

Assignment 5: The Event Organizer – Version 2

This version of the assignment is a Pass option, with the highest possible grade being a C. If you aim for a higher grade (A or B), you should complete Alternative 2 of this assignment, which is available on the same page and intended for those grades. In that case, you do not need to proceed with this version.

1. Objectives

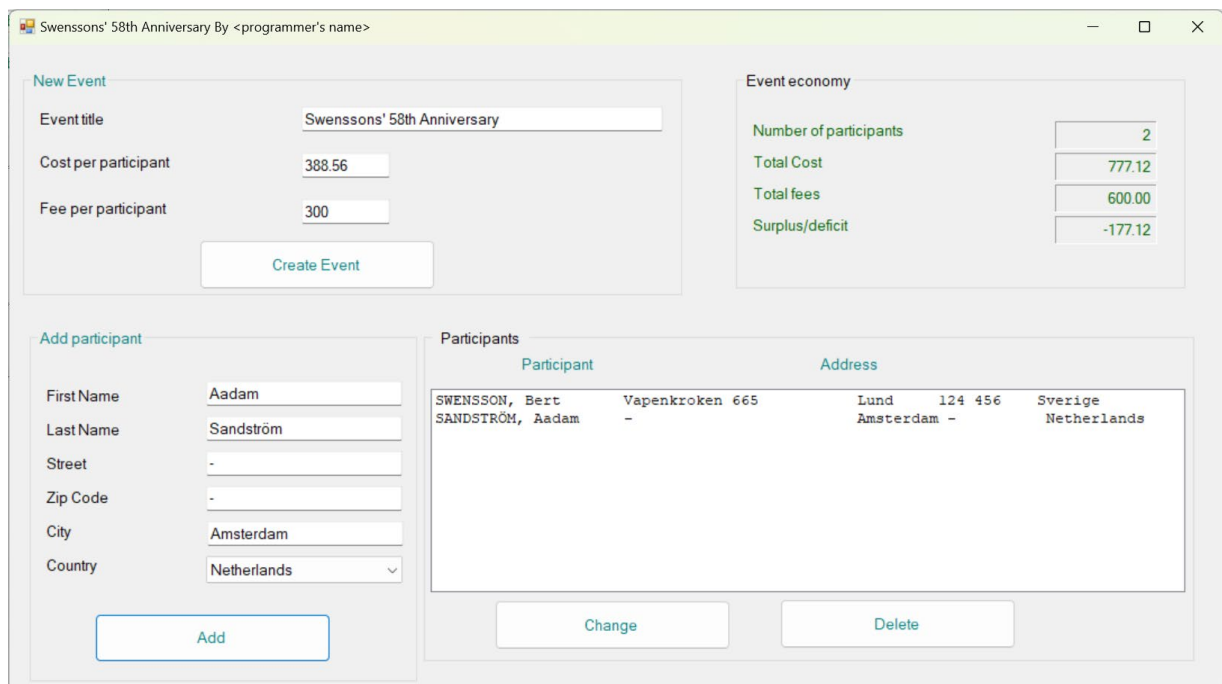
The main objectives of this assignment are to introduce you to the world of Object-Oriented Programming (OOP) by focusing on one of its essential aspects: **encapsulation**. We will also practice data hiding to fully achieve encapsulation. This assignment covers the following concepts:

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

2. Description

In the previous assignment, you developed an application to manage an event by creating a list of participants and storing their first and last names using a fixed-size array of strings.

In this assignment, we will expand the application with new features. We will introduce a **Participant** class to handle each participant's data, including their address. Additionally, we will create a **ParticipantManager** class to maintain and manage the list of participants. Below is an example of a potential GUI for input and output



The screenshot shows a GUI application titled "Swenssons' 58th Anniversary By <programmer's name>". The application is divided into four main sections:

- New Event:** Contains three input fields: "Event title" (with the value "Swenssons' 58th Anniversary"), "Cost per participant" (with the value "388.56"), and "Fee per participant" (with the value "300"). Below these fields is a "Create Event" button.
- Event economy:** Contains four rows of data, each with a label and a value in a box:

Number of participants	2
Total Cost	777.12
Total fees	600.00
Surplus/deficit	-177.12
- Add participant:** Contains six input fields: "First Name" (Aadam), "Last Name" (Sandström), "Street" (-), "Zip Code" (-), "City" (Amsterdam), and "Country" (Netherlands). Below these fields is an "Add" button.
- Participants:** Contains a table with two columns: "Participant" and "Address". The table has two rows of data:

Participant	Address
SWENSSON, Bert	Vapenkroken 665 Lund 124 456 Sverige
SANDSTRÖM, Aadam	- Amsterdam - Netherlands

 Below the table are two buttons: "Change" and "Delete".

Figure 1: A run-time example

3. Requirements for Grades D and C

- 3.1 All class fields must be declared as private. The use of public instance variables is strictly prohibited and will result in a required resubmission. To allow access to the private fields of a class, use properties.
- 3.2 Create a **Participant** class to store each participant's address, along with their first and last names. Use properties to access private fields and implement a **ToString()** method that returns a formatted string with the data saved in a Participant object.
- 3.3 Create an Address class to be used by the Participant class. The address should include details for the street, city, zip code, and country. Use the **string** data type for the first three fields, and an **enum** for countries. The **Countries** enum is available for download on the Assignment page in Canvas.
- 3.4 Create a **ParticipantManager** class that includes a List<T> collection to store Participant objects (i.e., List<Participant>). As a dynamic type, this collection automatically adjusts in size, so you don't need to set a maximum number of participants; it can grow and shrink as participants are added or removed.
- 3.5 Create an **EventManager** class that includes an instance of the **ParticipantManager** as a field, along with fields for cost per person and fee per person. The cost represents the budgeted amount the event owner will pay, while the fee is the amount each participant pays to the owner for participation.

All information about participants, such as the number of participants, should be provided by the **ParticipantManager** class. The total cost can be calculated as cost per person * number of participants, and the total fees as fee per person * number of participants.

- 3.6 The **MainForm** should contain only one instance variable, an instance of **EventManager**:

```
public partial class MainForm : Form
{
    EventManager eventManager;

    public MainForm()
    {
        InitializeComponent();
        InitializeGUI();
    }
    . . .
}
```

At program start, the group box for entering participant information should be disabled until the user has provided a title and clicked the "Create Event" button. If the user does not provide values for cost and fee per person, default to 0.0, but a title must be provided.

- 3.7 Display a list of registered participants in a ListBox, showing each participant's name and address details, as shown in Figure 1. Additionally, display information on the number of registered participants, total cost, and total fees on the GUI. The GUI should remain updated as data is added, modified, or deleted.

Requirements for a D Grade

3.8 The application should provide the following functionalities:

- Create a new event with a title, cost per person, and fee per person.
- Add a new participant with their name and address.
- Remove an existing participant.

3.9 Control must be done in your code so that the list is not indexed out of range.

Requirements for a C Grade:

In addition to the above requirements, implement the following:

- Modify data for an existing participant.

3.10 For the Change (Modify) and Delete buttons, the user must first highlight an item in the ListBox.

Optional (not mandatory): When displaying country names, replace any underscores (_) in names like United_States_Of_America with spaces, so it appears as "United States of America" for easier reading.

Optional (not mandatory): **Chain-Calling**: The Address class should include three constructors, with each constructor chain-calling another. Avoid duplicating code across constructors. Chain-calling means that a constructor with fewer parameters should call one with more parameters, passing along its own parameters plus default values as needed. For instance, a constructor with two parameters should call a constructor with three parameters, supplying the two values and a default for the third. If no such constructor exists, it should call the next available one with more parameters, and so on.

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

4. Submission, help and guidance

Compress your entire project, including all relevant files and folders, into a ZIP, RAR, or 7z format. Upload the compressed file to Canvas on the same page where you downloaded the assignment. Ensure that your project compiles without errors and that the program runs as expected.

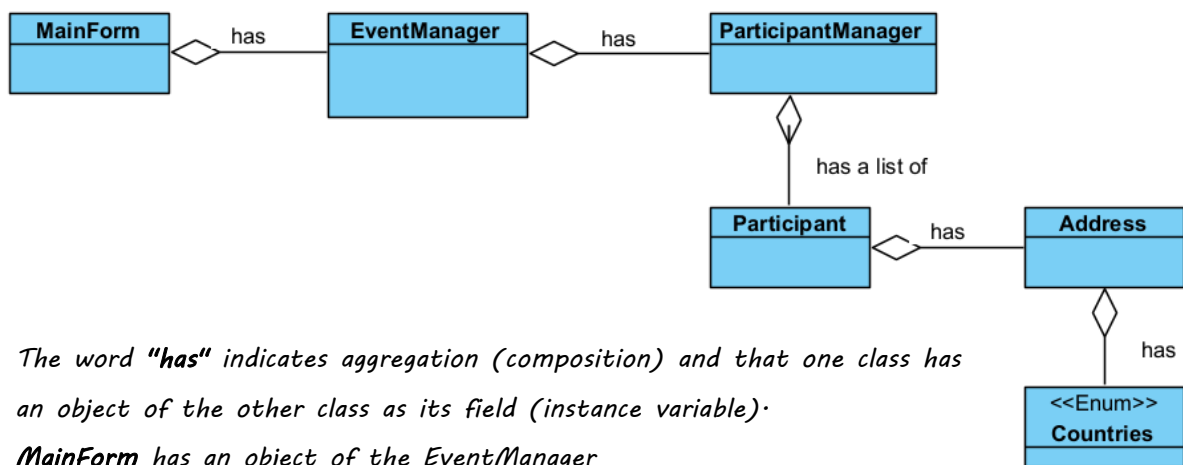
Note: To offer some guidance and ideas, the following sections provide hints and a number of class diagrams. These diagrams are optional; you are not required to follow them. You may design your GUI and organize your project and files according to your own preferences and judgment.

5. Help and Guidance

You may start by sketching your GUI form or by writing the classes needed to store and process data. Be sure to assign meaningful names to the controls (GUI components) in your design before programming the code-behind.

Once you're ready to start coding, it's essential to structure your solution thoughtfully by focusing on the necessary classes. In the example GUI above, the user is required to input details for both the event and the participants. This can provide insight into the classes you'll need to create, along with the data and methods that should be included in each related class.

The class diagram below illustrates the classes and their relationships.



The word "**has**" indicates aggregation (composition) and that one class has an object of the other class as its field (instance variable).

MainForm has an object of the **EventManager**

EventManager uses an object of **ParticipantManager**

ParticipantManager has a collection of **Participant** objects (1 to many)

Participant has an object of **Address**

Address uses the enum **Countries**.

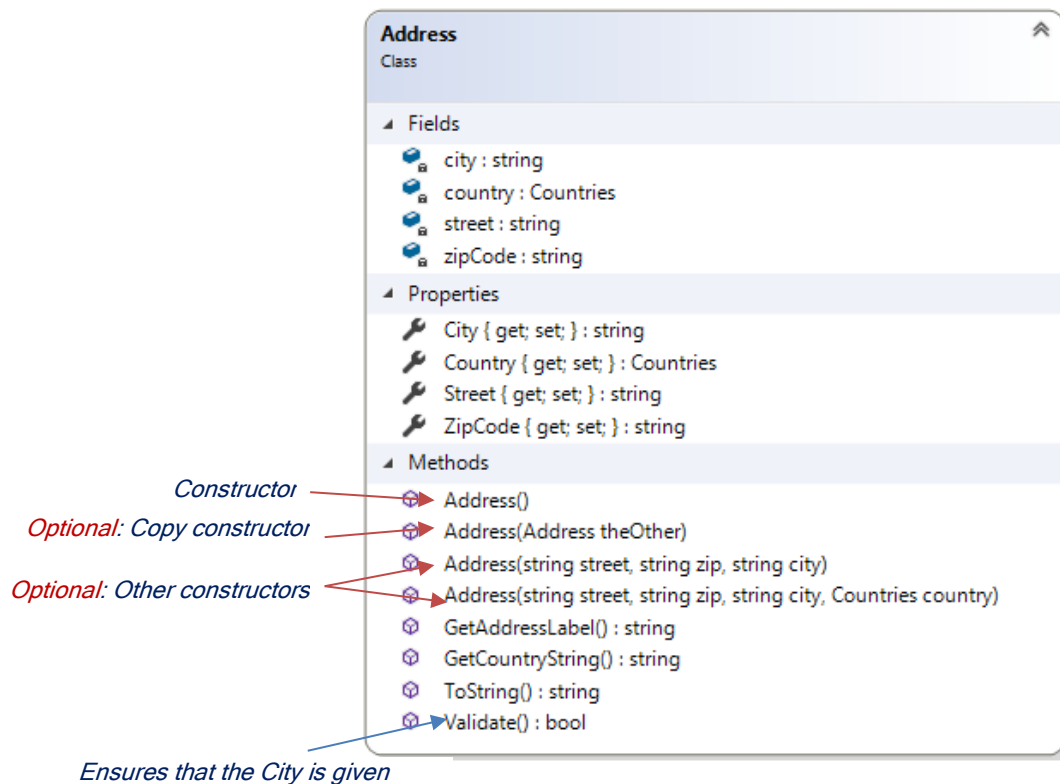
Get started with classes

When you start creating classes, begin with the class that has the fewest dependencies on others. In this case, the **Address** class is the least dependent, as it is an independent class. Next, develop the **Participant** class, which uses **Address**, followed by **ParticipantManager**, which relies on **Participant**. Once these classes are complete, create the **EventManager** class, which depends on **ParticipantManager**. Finally, write the code in **MainForm** to create and utilize an **EventManager** object.

Note: You do not need to implement all the methods shown in the class diagrams provided with the class descriptions below. If the purpose of a method isn't clear, feel free to skip it and implement your own solution instead. Write only those methods you need or create your own as necessary.

5.1. The Address class

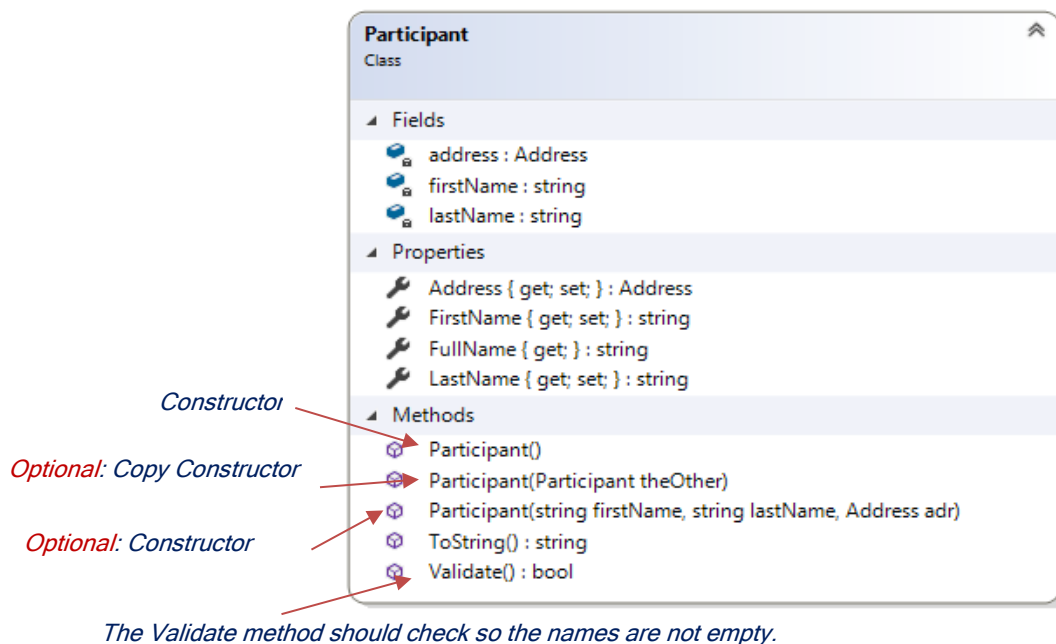
Create a new class **Address** and define the necessary fields within the class and implement properties to provide access to these fields.



The **Validate()** method should ensure that the City field, which is a string, is provided, meaning it must not be empty or null. You can verify any other string data using the following code:

```
bool ok = !string.IsNullOrEmpty(textToValidate);
```

5.2. The Participant class



5.3. The ParticipantManager class

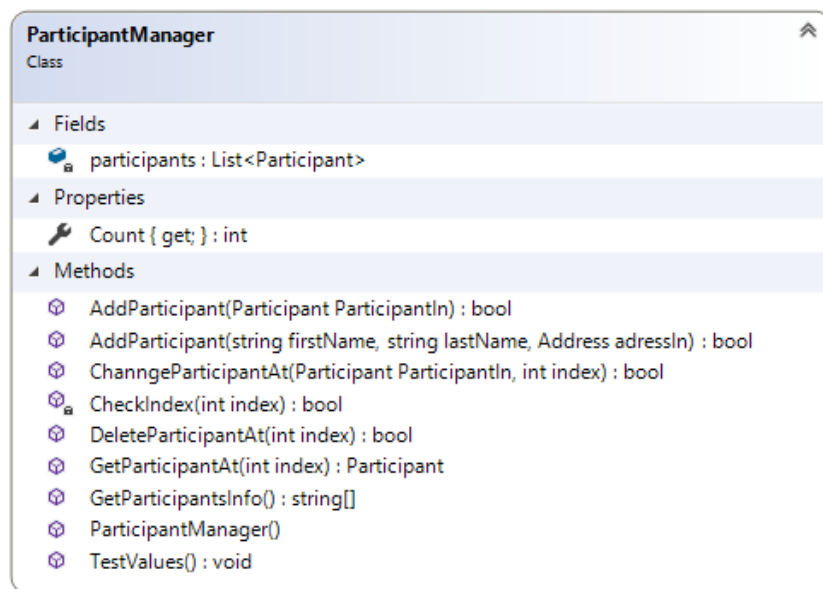
This class is responsible for maintaining and managing a collection of participants. Its functions include adding new participant objects, editing or deleting existing ones, retrieving a participant at a specified position, and returning a list of strings that represent each participant.

This class acts as a container for **Participant** objects. Use a **List<Participant>** to store the participants added through the user interface

```
class ParticipantManager
{
    //declare an object of the collection
    private List<Participant> participants;

    public ParticipantManager()
    {
        participants = new List<Participant>();
        //TestValues();
    }
}
```

The following class diagram can be used as a template.



The **GetParticipantsInfo** method returns an array of strings, where each string is the result of calling the **ToString()** method on a **Participant** object. In turn, the **ToString** method in the **Participant** class can also call the **ToString** method in the **Address** class

To store participant data, we will use a dynamic array, commonly referred to as a collection. Collections are a broad topic and will be covered in the next course, so for now, we'll focus on the commonly used and highly versatile **List<T>**. This generic collection works with all types of objects and is preferred over the older **ArrayList**, which is non-generic and less commonly used today.

Use a **List<Participant>** to store **Participant** objects in the **ParticipantManager** class, which will act as a container for managing customers

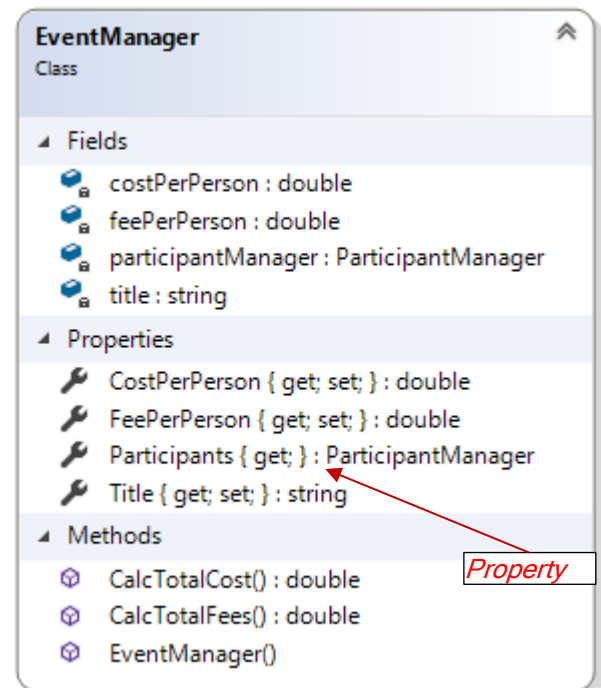
To prevent out-of-range indexing in the list, ensure that the index used to access an element is ≥ 0 and $<$ the list's length. This validation can be managed by a **CheckIndex** method, as shown in the figure above.

5.4. The EventManager class

Although the name **EventManager** might suggest it is a container class, it is not. While a name like Event could seem more fitting, we should avoid using it since Event is a reserved keyword in the .NET.

The primary responsibility of **EventManager** is to manage the event itself. As shown in the class diagram, it includes an instance of **ParticipantManager** (through aggregation) as one of its fields. **MainForm** should use an instance of **EventManager** to store not only event data, such as title, cost, and fee, but also to save participant data sent from MainForm to **ParticipantManager**.

In the class diagram, a property that returns the **ParticipantManager** is included. This property is necessary to allow MainForm access to the methods of **ParticipantManager**.



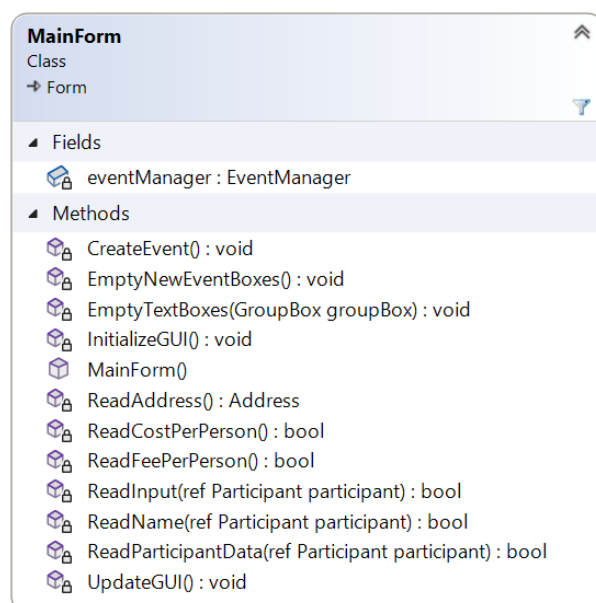
5.5. The MainForm class

MainForm should contain only one instance variable, which is an **EventManager** object. No additional instance variables should be declared.

The following class diagram for **MainForm** shows some of its class members. Event-handler methods and data related to GUI components are not included.

```
public partial class MainForm : Form
{
    EventManager eventManager;

    public MainForm()
    {
        InitializeComponent();
        InitializeGUI();
    }
}
```



Create an event. (Figure 1)

Create the instance variable for the event. Read the title, cost, and fee provided by the user, and store this data in the **EventManager** object by calling the corresponding properties.

Add a new participant

Create a new **Participant** object. Read and save the user input into this object, validate the input, and then add the object to the list of participants. Use the Participants property available in the eventManager object to accomplish this.:

```
eventManager.Participants.AddParticipant(participant);
```

For Grade C: Edit an existing participant:

- The user selects a participant from the ListBox, among the list of participants. If no participant is selected, display a message and cancel further processing.
- Using the selected index, retrieve the participant object from **ParticipantManager**.

```
eventManager.Participants.GetParticipantAt(..)
```

- Display the selected participant's data in the related input fields.
- When the user clicks the Edit button, update the participant information in **ParticipantManager** and refresh the GUI accordingly.

9.1 Delete an existing participant:

- The user selects a participant from the ListBox among the participants. If no participant is selected, display a message and stop further processing.
- Using the selected index, call the **ParticipantManager's** Remove (or Delete) method to remove the participant at that position in the list. Update the GUI accordingly.

Good Luck!

Farid Naisan,

Course Responsible and Instructor