

# **TME286**

## **Intelligent Agents**

---

### **Tutorial 2:**

### **Classes, class libraries, and generic lists**

Mattias Wahde  
2019

---

## 1. Introduction

In this tutorial we shall briefly introduce the concepts of classes and generic lists.

### 1.1 Preparation

Before proceeding, first complete Tutorial 1 and then read Sections A.2 and A.3 in the compendium!

## 2. Step-by-step tutorial

- Double-click on TutorialSolution.sln to open the TutorialSolution in Visual studio (see also Tutorial 1).
- Next, right-click on the icon in front of the text *Solution 'TutorialSolution'* at the top of the Solution Explorer. In the window that appears, left-click (once) on *Add* and then select *New Project*, as in Fig. 1.
- In the window that appears, check that the selected option (with gray background; see Fig. 2) is "Windows Forms App ..." (not "WPF App ..."). If not, click on "Windows Forms App ...". In the next step, change the Project name to "Tutorial2Application", and then click Create.

We have now added another application to the solution. In general, a solution can contain any number of projects, some of which are executable applications and other that are so called class libraries (containing code that is used by the executable applications). Very often, the applications have some connection to each other. Here, there is no real connection between the applications (that is, they would not run concurrently, communicating with each other etc.), other than the fact that they are both tutorials! Now, in order to run the (so far empty) Tutorial2Application one can right-click on Tutorial2Application in the Solution Explorer, then select *Debug – Start New Instance*. However, an easier way is to set Tutorial2Application as the **Startup Project**, i.e. the project that is started by default when the user presses the green arrow ("Start") at the top of the IDE window. In order to do so, right-click on Tutorial2Application in the Solution Explorer and select *Set as Startup Project*. Then either press the green arrow or F5. The square-shaped Form1 should appear.

Stop the program by selecting *Debug – Stop Debugging* in the top menu of the IDE window or by pressing Shift – F5. To stop execution, one can also click on the cross in the upper-right corner of Form1. However, that action is not *certain* to stop execution of a program: It does close the form permanently, but if the program has one or several threads running, it will not fully stop until those threads finish execution. Thus, to make absolutely sure that execution has stopped, it is better to use *Debug – Stop Debugging* as described above. Note that if one wants to edit the code and then compile again (to build a new executable file), the program must be fully stopped, otherwise Windows will prevent the executable file from being overwritten by the new version.

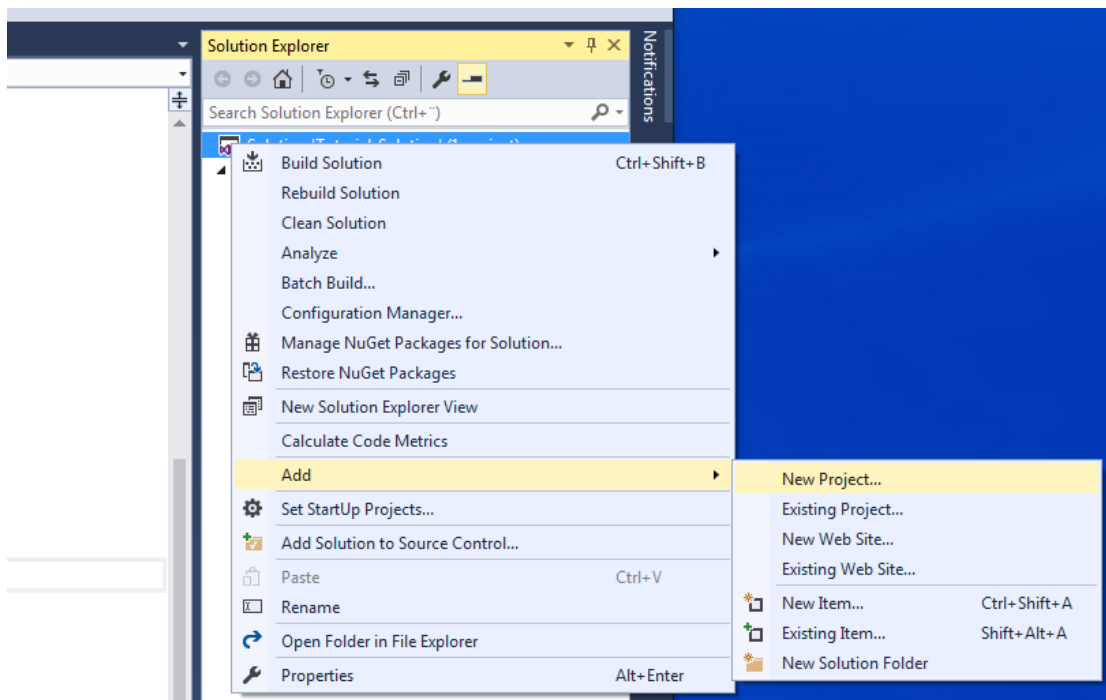


Fig. 1: Adding a new project.

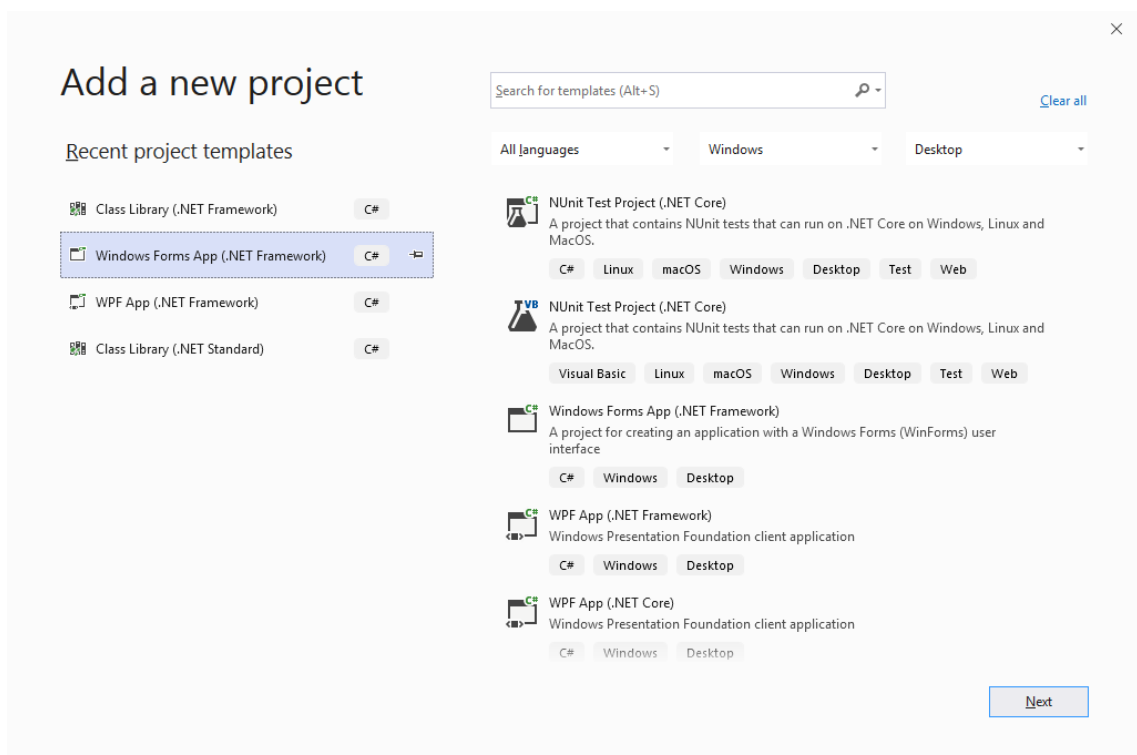


Fig. 2: Naming the new project.

- Now, rename the form (Form1) as “Tutorial2Form”, by right-clicking on the Form in the Solution Explorer, selecting Rename, and typing in the new name. Make sure that the name ends with “.cs”. Then press Return. You will be asked by C# if you want to rename all references to Form1. Click “Yes”. By doing so, one makes sure that C# automatically makes the appropriate changes throughout the entire application (or the entire solution, when applicable).

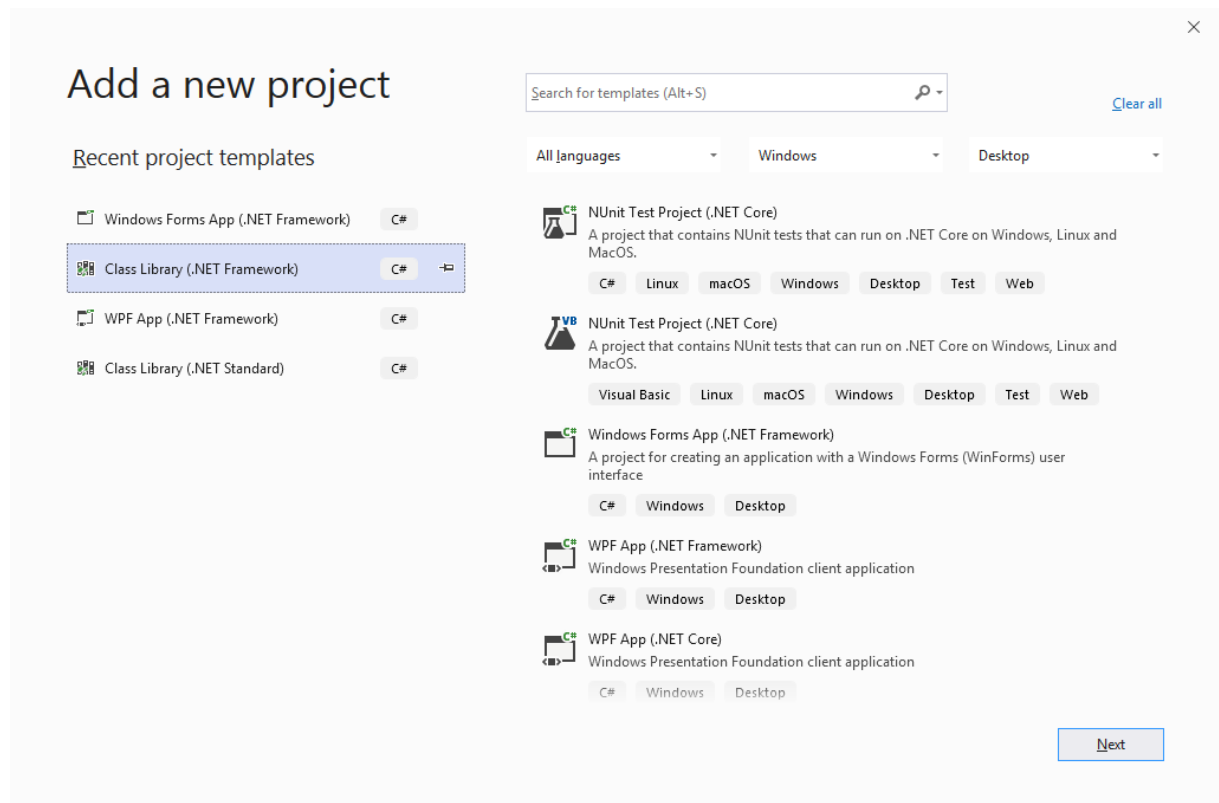


Fig. 3 Adding a class library.

## 2.1 Adding classes (in a class library)

- C# is an object-oriented language in which data structures known as classes are used. In an informal sense, a class contains various pieces of information (**fields**) that belong together, as well as various ways of manipulating the information (**methods**). As a simple example, consider a class that is intended to store information about countries. There are various pieces of information that could be relevant, such as the name of a country, its population, its major cities, the languages spoken etc. etc. All that information can be encapsulated in a class, as we shall see.
- A set of classes that belong together typically forms a **class library**. While it is certainly possible to add classes directly in an application (for example the Tutorial2Application), it is often better to collect classes in a class library *if* those classes are to be used later, in a different application. Then, one can simply include the class library in that other application, and thereby get direct access to all the classes. An example should clarify these concepts:

In the Solution Explorer, right click (again) on the icon in front of *Solution 'TutorialSolution'*. However, this time, instead of "Windows Forms Application" select "Class Library" (see Fig. 3) and click *Next*. In the next step, change the name to *GeographyLibrary* and click *Create*.

Fig. 4. The first stages of the Country class.

- As you can see, the GeographyLibrary, containing one empty class (Class 1), has been added to the solution (see the Solution Explorer).
- Next, rename Class1 (in the GeographyLibrary) as “Country” (still with the suffix “.cs”, of course); If you forgot how to do that, see the renaming of the form at the bottom of p. 2.
- Double-click on Country.cs in the Solution Explorer. The code for the class appears in the Code Editor. Now, add code as shown in Fig. 4 above. We have added two **fields**, *name* and *population*, which are marked as **private**, meaning that those fields are visible to any methods in the class, but not to other classes (nor, indeed, to other instances of the Country class). In many cases, one wants to make information available so that it can be manipulated by other classes. It is possible, but not recommended, to make the fields as **public**. However, a better approach is to separate a field (which is used internally in the class) from a **property** (which can be accessed from outside the class). This is so since, very often, the programmer wants to control the actions that may be applied to the fields of a given class. For example, one can make a property which makes it possible to read the value of a field, but not to change it etc. It is also possible to put code (e.g. event handlers) in a property definition, so that some action is taken whenever the property is accessed from outside the class. Here, however, the properties will be used in a simple fashion, just getting or setting the corresponding field value.

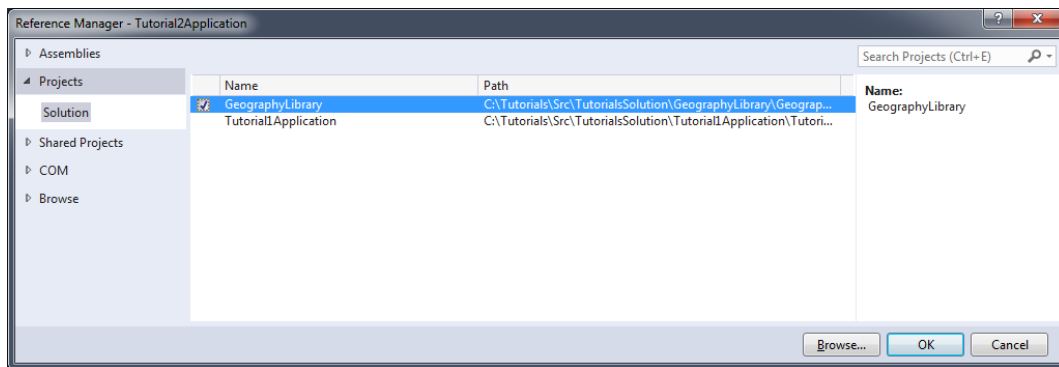


Fig. 5 Adding a reference to the GeographyLibrary.

**Note:** As per our coding standard, fields always begin with lowercase letters, and properties with uppercase letters.

- Let us now use this simple class in our application. Here, a small problem occurs: We have not yet told our application (Tutorial2Application) where it can find the Country class. In order to be used by an application a class library must first be referenced. Do the following: In the Solution Explorer, right-click on References under the Tutorial2Application. Then select *Add Reference..* Now, the window that appears (Fig. 5) may look a bit different for different versions of C#. In any case, select *Projects* and then tick “GeographyLibrary”, and click OK.

The Tutorial2Application is now aware of the GeographyLibrary, but in order to access its contents in the most convenient way, one should also add the GeographyLibrary in the using clauses in the main form of the Tutorial2Application. Thus, right-click on “Tutorial2Form” in the Solution Explorer, select “View code”. At the end of the using clauses, after the line `using System.Windows.Forms;` add the line `using GeographyLibrary;` (see also Fig. 7 below).

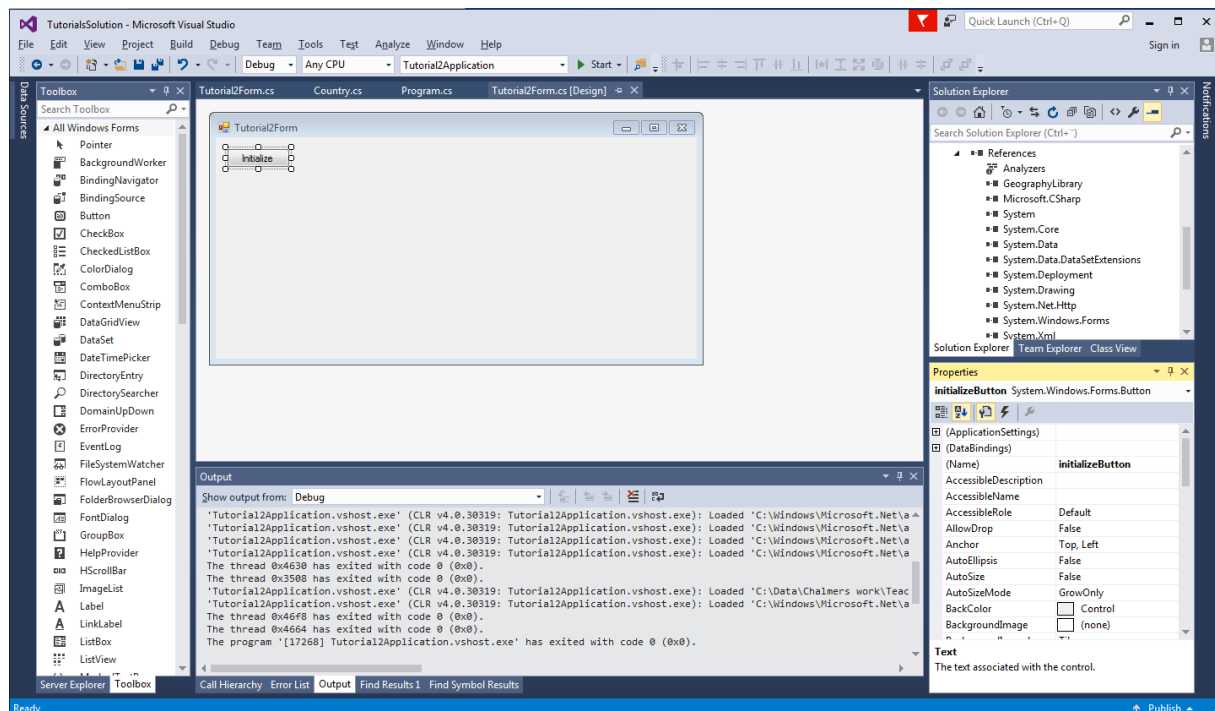
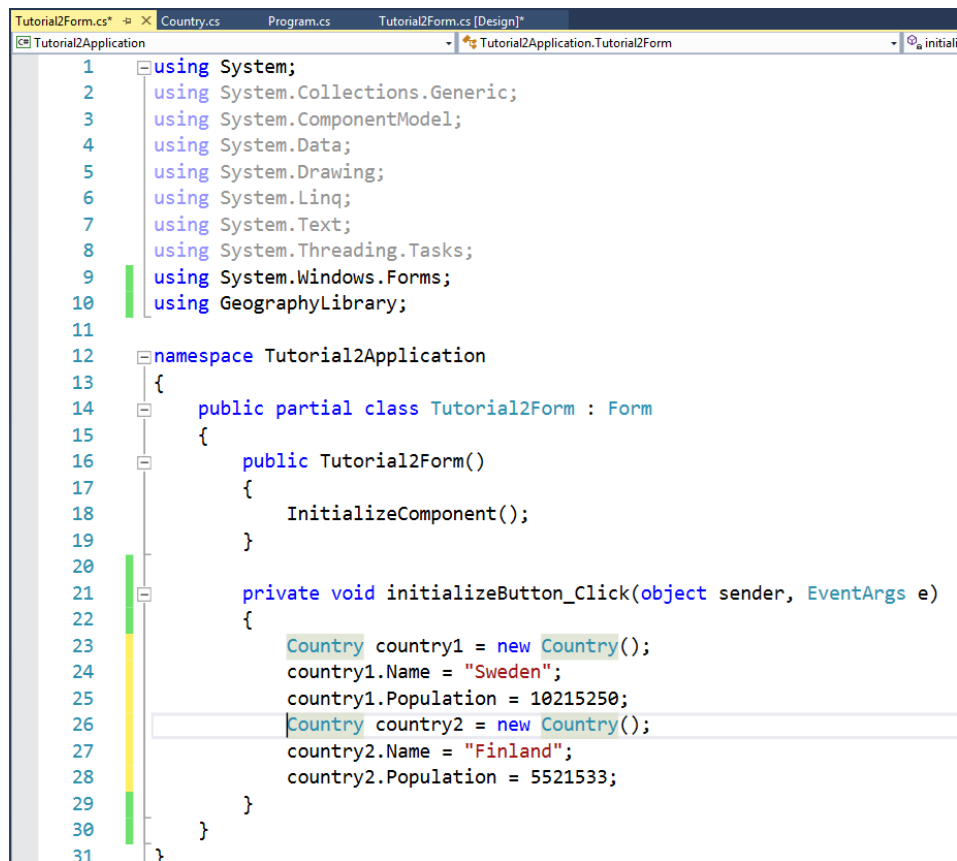


Fig. 6 The appearance of the Tutorial2Form after adding the initializeButton.



```

1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel;
4  using System.Data;
5  using System.Drawing;
6  using System.Linq;
7  using System.Text;
8  using System.Threading.Tasks;
9  using System.Windows.Forms;
10 using GeographyLibrary;
11
12 namespace Tutorial2Application
13 {
14     public partial class Tutorial2Form : Form
15     {
16         public Tutorial2Form()
17         {
18             InitializeComponent();
19         }
20
21         private void initializeButton_Click(object sender, EventArgs e)
22         {
23             Country country1 = new Country();
24             country1.Name = "Sweden";
25             country1.Population = 10215250;
26             Country country2 = new Country();
27             country2.Name = "Finland";
28             country2.Population = 5521533;
29         }
30     }
31 }

```

Fig. 7: Generating two instances of the Country class.

- We will now start using the Country class. Open the designer view (for the Tutorial2Form) by double-clicking on Tutorial2Form in the Solution Explorer. In order to make the form a bit nicer, extend it a bit to the left, and change (in the Property panel; see also Tutorial 1) its Text property from "Form1" to "Tutorial2Form".

Then, drag a button (from the Toolbox; see also Appendix A.1 in the compendium) onto the form, and change its name property to "initializeButton" and the Text to "Initialize". (You may need to click once on the button first, but do *not* double-click! If you accidentally double-click, remove the event handler that then appears, as described at the very end of Tutorial 1). The form should now appear as in Fig .6.

- Next, double-click on the initializeButton (in the Windows Forms Designer in the IDE). C# will then automatically write skeleton code for the click event handler (see also Tutorial 1).
- Start by writing code as in Fig. 7. With this simple code, an **instance** of the Country class, named "country1" (lowercase letters) is generated, and then another instance named country2. Of course, any valid C# name can be used for the instances, in principle, but note that we should always follow the coding standard and give clear, descriptive names (that, in the case of fields, start with a lowercase letter).

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace GeographyLibrary
8  {
9      public class Country
10     {
11         private string name;
12         private int population;
13
14         public Country(string name, int population)
15         {
16             this.name = name;
17             this.population = population;
18         }
19
20         public string Name
21         {
22             get { return name; }
23             set { name = value; }
24         }
25
26         public int Population
27         {
28             get { return population; }
29             set { population = value; }
30         }
31     }
32 }

```

Fig. 8: A parameterized constructor for the Country class.

Now, in order to speed up the process of defining instances, one can write a **parameterized constructor** for the class. In general, every class has a **default constructor** with the same name as the class, and without any input parameters; see Lines 23 and 26 in Fig. 7. This constructor simply generates an instance of the class. Open the Country.cs class again (by double-clicking on it in the Solution Explorer). Then add the parameterized constructor (Lines 14-18 in Fig. 8 above). The specification “this” is used to distinguish between a field in the current instance of Country (e.g. this.population) from the input parameter (population) with the same name<sup>1</sup>.

- Next, change the code in the initializeButton\_Click method of Tutorial2Application as in Fig. 9 below. Note that, now, an instance of the Country class can be generated and fully specified with a single line of code.

```

private void initializeButton_Click(object sender, EventArgs e)
{
    Country country1 = new Country("Sweden", 10215250);
    Country country2 = new Country("Finland", 5521533);
}

```

Fig. 9: The modified initializeButton\_Click method.

<sup>1</sup> Some coding standards require that input parameters should be prefixed by some character e.g. “\_”. In our coding standard (which *must* be followed in the assignments) this is *not* the case. We use the same name for a parameter and a field, and distinguish them instead via the “this” keyword, as explained above.



```

namespace Tutorial2Application
{
    public partial class Tutorial2Form : Form
    {
        private List<Country> countryList;
    }
}

```

Fig. 10: Adding the countryList field, just below the line defining the form.

```

private void initializeButton_Click(object sender, EventArgs e)
{
    countryList = new List<Country>();
    Country country = new Country("Sweden", 10215250);
    countryList.Add(country);
    country = new Country("Finland", 5521533);
    countryList.Add(country);
    country = new Country("Norway", 5295519);
    countryList.Add(country);
}

```

Fig. 11: Instantiating a generic list of Country instances.

**Note:** If one still wants to have the opportunity to use the non-parameterized constructor (for the Country class), one has to add code for it, by adding the empty method `public Country() { }` in the Country class. Of course, it is possible to add code in this constructor as well, if desired.

- The code above does generate two instances, country1 and country2, but the variables are **local**. That is, they are only visible in the method in which they are defined. Also, if one wants to add a list of countries, it would be better to define such a list, rather than generating instance names such as country1, country2 etc. This is our next step.

## 2.2 Generic lists

- Thus, now *remove* the two lines of code (in the initializeButton\_Click method) that generate country1 and country2 (but do not remove the method, just its contents!). Next, just below the line `public partial class Tutorial2Form:Form`, add the line `private List<Country> countryList;` (with capital L in “List” and capital “C” in “Country” (but not in “countryList”). Do not forget the semi-colon “;” at the end of the line; see also Fig. 10 above). Then, in the initializeButton\_Click method, add code as in Fig. 11.

This code generates an example of what is known as a **generic list**. In this case, the list contains instances of the Country class, but such a list can contain instances of any class (hence the name “generic list”). Now, C# contains a large set of methods for manipulating such lists, some of which (there are many more!) will be exemplified below.

As can be seen in Fig. 11, one must call a constructor to set up the list (countryList). Next, an instance of Country is generated and is added to the list. Note that, here, there is no need to use separate names for the various instances of Country, *provided* (note!) that one calls the constructor for every new country that is being added: When the constructor is called, C# allocates a separate space in memory for the instance and all its fields.

Thus, it would be *wrong* to write the code as in Fig. 12: In this case, only a single instance of Country would be generated, and three identical copies would be included in countryList. In reality, a generic list contains a set of pointers to instances, and in Fig. 12 there is only one instance of Country. Thus, after the last line, the list will contain three elements, all of which will have the values name = Norway, population = 5295519. It is a good idea to put a breakpoint at the last line of the method (both for the code in Fig. 11 and the incorrect code in Fig. 12) and then check the contents of the list; see also the Section A.3 of Appendix A in the compendium.

```
private void initializeButton_Click(object sender, EventArgs e)
{
    countryList = new List<Country>();
    Country country = new Country("Sweden", 10215250);
    countryList.Add(country);
    country.Name = "Finland";
    country.Population = 5521533;
    countryList.Add(country);
    country.Name = "Norway";
    country.Population = 5295519;
    countryList.Add(country);
}
```

Fig. 12 An (note!) **incorrect** way of specifying the list.

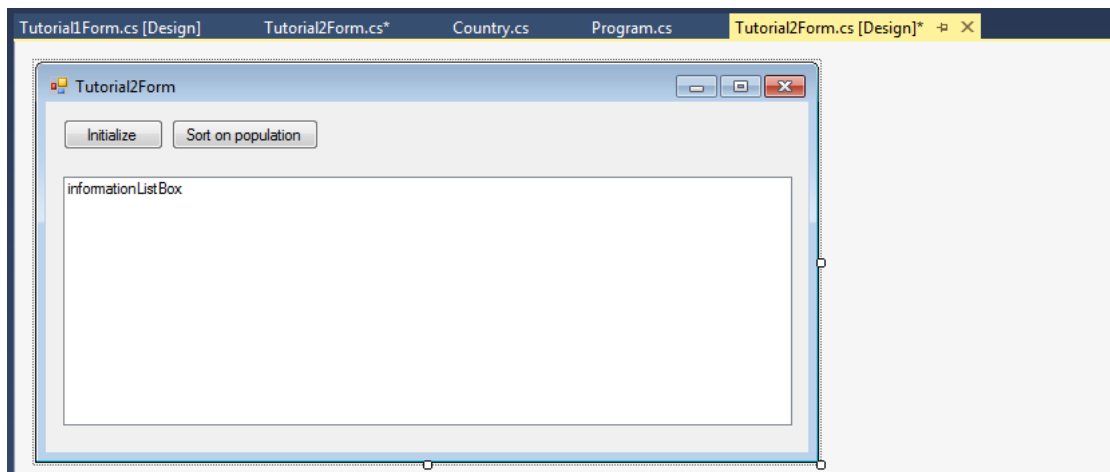


Fig. 13 The form after adding a Listbox and another button.

- Now, restore the code so that it again looks as in Fig. 11. Once a generic list has been defined, there are many ways of manipulating its contents. Before doing so, we will add a bit of code for displaying the result of the various operations. Thus, from the ToolBox (in the IDE), drag a Listbox onto the form. Then change its name to informationListBox. Next, add another button to the form, and set its Name property (in the Properties panel) to sortButton and its Text property to *Sort on population* (Drag the right corner of the button a bit, so that one can see the whole text, not just “Sort on”). Finally, set the Enabled property (of the sortButton) to false. The form should now look as in Fig. 13.

**Note:** If one wants to be able to rescale the form and having a GUI component (for example a Listbox) rescale with it, one can set the Anchor properties (Top, Left, Right, Bottom, as desired) of the GUI component in question. An even better way is usually to Dock the component, but we cannot do so here without hiding the button.

- The reason for setting the Enabled property of the sortButton to `false` is that one cannot apply any operation on the countryList before it has been instantiated. Thus, at the very end of the initializeButton\_Click method, add the line `sortButton.Enabled = true;` With this modification, the sortButton will be enabled only once the countryList has been instantiated (by a click on the initializeButton).
- Now, double-click on the sortButton (in the Windows Forms Designer in the IDE), so that C# can write the skeleton code for the button's click event handler. Then add the code in Fig. 14 and run the program. Click first on *Initialize* and then on *Sort on population*.

```
private void initializeButton_Click(object sender, EventArgs e)
{
    countryList = new List<Country>();
    Country country = new Country("Sweden", 10215250);
    countryList.Add(country);
    country = new Country("Finland", 5521533);
    countryList.Add(country);
    country = new Country("Norway", 5295519);
    countryList.Add(country);
    sortButton.Enabled = true;
}

private void sortButton_Click(object sender, EventArgs e)
{
    countryList.Sort((a, b) => a.Population.CompareTo(b.Population));
    informationListBox.Items.Clear();
    foreach (Country country in countryList)
    {
        string countryInformation = country.Name + ": population = " + country.Population;
        informationListBox.Items.Add(countryInformation);
    }
}
```

**Fig. 14 The two click event handlers after adding code as described above.**

- As can be seen in the output, the information about the countries is displayed in the ListBox on the form, in ascending order of population. The sortButton\_Click method does two things: First, it sorts the list of countries. Next, it prints the information to the informationListBox.
- By default, the sort order is *ascending*. For reverse sorting (descending order), add the line `countryList.Reverse();` after the line that does the sorting, or use the `SortDescending()` method.
- Now, the syntax for the sorting may appear a bit complex at first. In fact, for a list consisting of simple types, such as `int` or `double`, one can simply call the `Sort()` command, without any further specification, in order to get the list sorted in ascending order. As an example, try generating a `List<Int>` (for instance in the click event handler of another button that you can now add to the form, using the procedure outlined above) as in Fig. 15.

```
private void IntegerSortButton_Click(object sender, EventArgs e)
{
    List<int> integerList = new List<int>();
    integerList.Add(5); // => {5}
    integerList.Add(3); // => {5,3}
    integerList.Add(11); // => {5,3,11}
    integerList.Sort(); // => {3,5,11}
}
```

**Fig. 15: Sorting a generic list with simple type elements (in this case *int*).**

- Now, returning to the example with the `countryList` (Fig. 14), the reason for the somewhat complex syntax is that there is no default way to sort instances of `Country`. For example, one might as well sort them by name (in alphabetical order, or based on the length of the name, or in some other way) rather than by population. Thus, one must specify, as in done in Fig. 14, exactly *how* the sorting should be carried out, in this case by comparing population values.

```
private void sortButton_Click(object sender, EventArgs e)
{
    countryList = countryList.FindAll(c => c.Population < 10000000);
    countryList.Sort((a, b) => a.Population.CompareTo(b.Population));
    informationListBox.Items.Clear();
    foreach (Country country in countryList)
    {
        string countryInformation = country.Name + ": population = " + country.Population;
        informationListBox.Items.Add(countryInformation);
    }
}
```

**Fig. 16: An illustration of the `FindAll()` command for generic lists.**

- Generic lists make it possible to carry out many different operations on lists, not just sorting. For example, one such operation finds elements (or their indices in the list) that fulfil some criterion. An example is given in Fig. 16 where now the sorting is preceded by an extraction of all countries whose population is smaller than 10 million.
- In this example, the `countryList` is overwritten (that is, countries with population of 10 million and above are removed). It is of course possible to generate a new list as well (if one wants to keep the `countryList` intact) as

```
List<Country> newCountryList = countryList.FindAll(c => c.Population < 10000000);
```

and then instead sort the new list and print the corresponding information. Note that the new list will contain pointers to the selected instances of the `Country` class, *not* new instances of the `Country` class.

**Note:** Numerical parameters (in this case 10000000) should not really be hard-coded as in Fig. 16, but should instead be parameterized (i.e. by defining a variable `populationThreshold` and setting its value to 10000000) or, if the parameter really is not to be changed, defining a constant (here: `POPULATION_THRESHOLD`); see the coding standard. In this case, one can certainly imagine that the user may wish to change the threshold for the sorting (for example

from 10 million to 20 million). However, for the purpose of illustrating the FindAll() command, this minor violation of the coding standard can perhaps be accepted here.

- Other useful generic list commands that you should study (there are many examples in the source code provided on the course web page and on the internet, and you are of course welcome to ask Mattias or Minerva any time) are Find(), FindIndex(), Sum(), Reverse(), Remove(), RemoveAt(), Where() etc. As an example, the following line will compute the total population of all the countries in countryList: `int totalPopulation = countryList.Sum(d => d.Population)`.