

Assignment 4: Properties and Arrays

APU'S COOK BOOK

1. Objectives

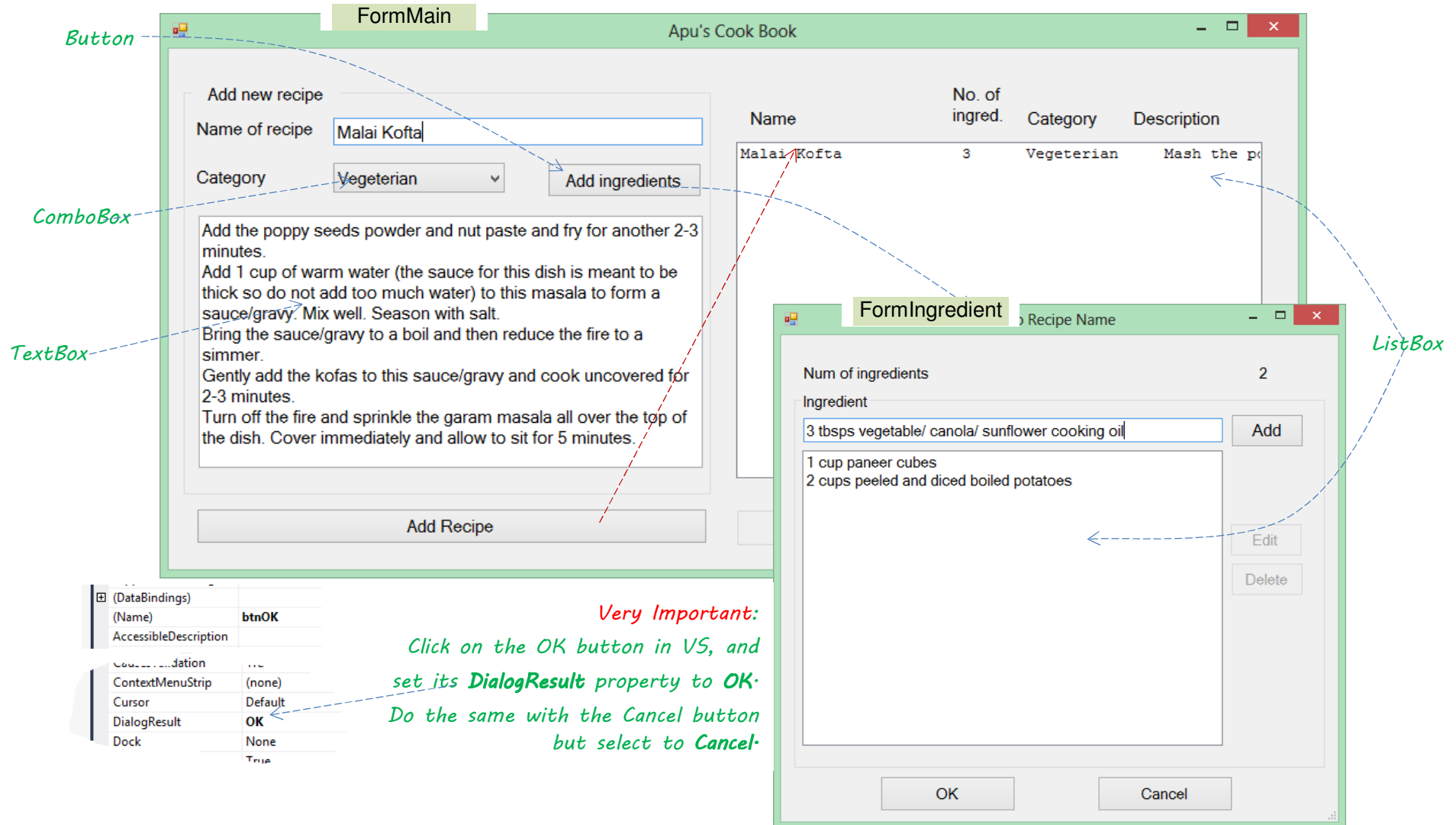
The main objective of this assignment is to work with one-dimensional arrays and learn how to write and use properties to access values saved in the private fields of an object.

Arrays are used to save a list of values of a single type as well as a list of objects. Moreover, this assignment gives further training in using constructors, properties, methods and enums. In this assignment, you will also learn to use multiple forms and make them communicate with each other. However, just as with other objects, there will be a client-server (giving service, receiving service), relation that will be implemented. Circular calls between two objects is not a good way and is to be avoided. By a server class I mean a class that is used by a client class. To make this clear, suppose that **FormMain** has a field (instance variable) **currRecipe** which is an object of a class called **Recipe**. The association is called “has a” and **FormMain** becomes a client class while **Recipe** is a server class. This is in no way any standard definition; I use them here only to clarify my point of view. The Client-Server term is commonly used with web, data base and network programming and that is not what is meant here

Restriction: To make sure that everyone works with ordinary arrays, **use of collections is not allowed** in this assignment. Collections of type List (and ArrayList) is implemented in the next assignment.

2. Description

In this assignment, we make a Recipe Book for use in Apu's restaurant. In this version, we will save the recipes in an array in the memory of the computer (in the program), not to any hard disk or database. We will learn to accomplish data persistence in the next C# course. An example of the design of the GUI for application is given on the next page. When adding a new recipe, the user writes the name of the recipe in the TextBox, selects a type from the ComboBox list (Vegetarian selected in the example), writes instructions in the large TextBox and clicks the button **Add ingredients** to input ingredients. For this part, a new Form (**FormIngredients**) is displayed, where the user uses the TextBox to specify a text for an ingredient, clicks the **Add** button and the program shows the input in the ListBox. After closing this form, the user can click the **Add Recipe** button and then **FormMain** saves the recipe in the **recipeMngr** and updates the ListBox (at the right side of the first Form).



FormMain

Add new recipe

Name of recipe:

Category:

Add ingredients

Add the poppy seeds powder and nut paste and fry for another 2-3 minutes.
Add 1 cup of warm water (the sauce for this dish is meant to be thick so do not add too much water) to this masala to form a sauce/gravy. Mix well. Season with salt.
Bring the sauce/gravy to a boil and then reduce the fire to a simmer.
Gently add the kofas to this sauce/gravy and cook uncovered for 2-3 minutes.
Turn off the fire and sprinkle the garam masala all over the top of the dish. Cover immediately and allow to sit for 5 minutes.

Add Recipe

FormIngredient

Recipe Name

Num of ingredients: 2

Ingredient

3 tbsps vegetable/ canola/ sunflower cooking oil

1 cup paneer cubes

2 cups peeled and diced boiled potatoes

Edit

Delete

OK

Cancel

Very Important:

Click on the OK button in VS, and set its DialogResult property to OK. Do the same with the Cancel button but select to Cancel.

(DataBindings)	
(Name)	btnOK
AccessibleDescription	
AutoComplete	
ContextMenuStrip	(none)
Cursor	Default
DialogResult	OK
Dock	None
Enabled	True

Every recipe has:

- a list of ingredients (strings)
- a description (instructions)
- a category (use any grouping, ex Meat, Fish, Sea Food, Vegetarian).

A screen image of the project structure shown to the right and a class diagram shown below should give you an idea of the whole application.

Category is an enum defining food categories.

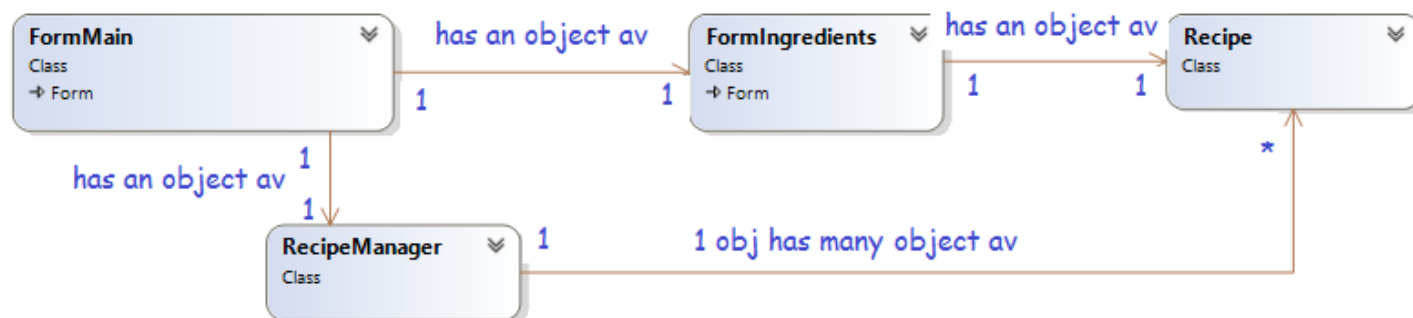
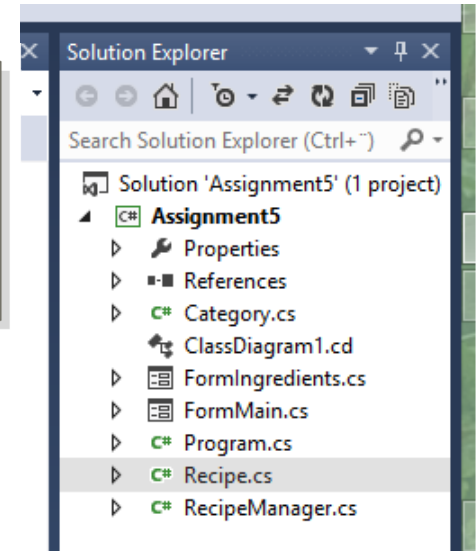
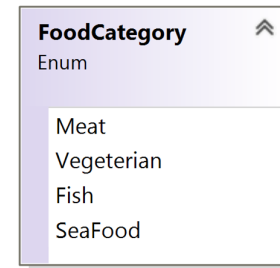
FormMain is the starting object. It uses an object of the **FormIngredients** class to provide an interface for inputting ingredients, as shown in the run example images above.

Recipe is a class that defines a recipe with name, category, ingredients and a description.

RecipeManager defines an array of recipes.

The ingredients are saved in an array of strings in the **Recipe** objects.

The recipes are saved in an array with elements of the type **Recipe** in the class **RecipeManager**.



3. Features and requirements for a Pass (C) grade

Write an application that has the following features:

- Saves data for a recipe as described above in an object (write the class **Recipe**).
- Saves a number of recipes in another object (write the class **RecipeManager**).
- Displays forms to the user to give recipe details (write the classes **FormMain** and **FormIngredients**).
- The forms and controls (such ListBox, TextBox, etc) should only display data taken from other objects in the application (Recipe, RecipeManager, FoodCategory, etc.).

The maximum number of ingredients for a single recipe as well as the maximum number of recipes that the application can save can be hardcoded in the **FormMain** as two constants. These values should then be passed to the Recipe and the RecipeManager when calling their constructors. Suggested max values:

Max number of ingredients = 50;

Max number of recipes = 200

The user should be able to give a name, a description, and a category type to define a recipe. She should also be able to specify a list of ingredients which can be saved as a list of strings. An ingredient may be defined as the user wishes ("2 dl low fat milk"). The program should then save the recipe in an object of the **RecipeManager**.

In case you have enough experience of Windows Forms, you may try working with Windows Presentation Foundation (WPF).

- 3.1 The Add button must work well in both of the forms, to add ingredients to the recipe being inputted and to add the recipe to the lists of recipes in the **RecipeManager**.
- 3.2 Use an array of strings in the **Recipe** class for storing ingredients. The size of the array (number of elements) should be set by the client object creating an instance of the **Recipe** class. This means that the **Recipe** class should provide a constructor with one parameter for the max number of ingredients.
- 3.3 Use an array with element of the type **Recipe** in the class **RecipeManager**. The size of the array is set through an argument in the constructor just as explained above for the **Recipe** class.

4. Features and requirements for a higher (A, B) grade

- 4.1 When the user clicks on an item in the ListBox on **FormMain**, the data fields in the left part of the GUI should be filled with data from the selected item.
- 4.2 Draw two buttons, **Edit** and **Delete**, on the **FormMain**. The user should be able to change the data for the selected item in the ListBox in the **FormMain** when pressing the Edit button. The GUI should then be updated accordingly.
- 4.3 When the user presses the **Delete** button, the selected item in the ListBox should be deleted from the array in the **RecipeManger** and the GUI should be updated. By deleting it is to be understood that the corresponding element is to be set to **null** or marked in other ways so you know the position is empty.
- 4.4 The **Edit** and **Delete** buttons on the **FormIngredients** should also work properly.

5. Functional Requirements (both C and AB grades)

- 5.1 The application should work well and not crash or generate exceptions due to invalid user input or bugs in the program.
- 5.2 The user should select an option from the list in the ComboBox and not be able to write a text in the textbox part of the control. This can be achieved by making the **ComboBox** readonly; set the **DropDownStyle** property of the ComboBox to **DropDownList**.

6. Structural Requirements (both C and AB grades)

7. General:

- 7.1 All general quality requirements and recommendations in the previous assignment is valid for this assignment as well.
- 7.2 All instance variables (fields) are to be private. Public fields are strictly forbidden.

- 7.3 Use **properties** to access private variables.
- 7.4 Use short methods with one task per method.
- 7.5 Document your methods and classes by writing comments.

If you would like to apply your own solution, have the above requirements in mind and as long as you maintain a good code quality, there will be no problem in grading your assignment. In case of unsatisfactory results, you will get a chance to do complementary work and resubmit.

8. Some quick help

Glossary:

Client class: A class that uses another class, e.g. **FormMain** becomes a client class to **RecipeManager** which in turn is a client class to Recipe class.

FormMain may also use a Recipe object (**currRecipe** in the example given earlier) and thus becomes a client class to Recipe.

The Recipe class is a server class to both the **RecipeManager** and **FormMain**.

Argument: values sent to method parameters: Values that are sent to a method by a caller method are also known as “Actual Parameters” or “**Actual Arguments**”. Method parameters (in the callee method) are also known as “Formal Parameters” or “**Formal Argument**”.

Constructor call:

```
private Recipe currRecipe = new Recipe(maxNumOfIngredients);
```

Call to Recipes constructor

The variable **maxNumOfIngredients** is an argument sent to the constructor. The whole statement creates an object of Recipe and the object is available through the reference variable **currRecipe**.

- 8.1 The class specifications below might give you an idea of the methods you may need to write in your classes. If you can't figure out what each method is intended for, check the help document or just ignore them and write your own methods to make things work.

Recipe
 Class

Fields

- category : FoodCategory
- description : string
- ingredientArray : string[]
- name : string

Properties

- Category { get; set; } : FoodCategory
- CurrentNumOfIngredients { get; } : int
- Description { get; set; } : string
- Ingredients { get; set; } : string[]
- MaxNumOfIngredients { get; } : int
- Name { get; set; } : string

Methods

- AddIngredient(string value) : bool
- ChangeIngredientAt(int index, string value) : bool
- CheckIndex(int index) : bool
- DefaultValues() : void
- FindVacantPos() : int
- Recipe(int maxNumOfIngredients)
- ToString() : string

RecipeManager
 Class

Fields

- numberOfItems : int
- recipeList : Recipe[]

Properties

- NumOfItems { get; } : int

Methods

- Add(Recipe newRecipe) : bool
- Add(string name, string[] ingredients) : bool
- GetIngredients(int index) : string[]
- RecipeListToString() : string[]
- RecipeManager(int maxNumOfElements)

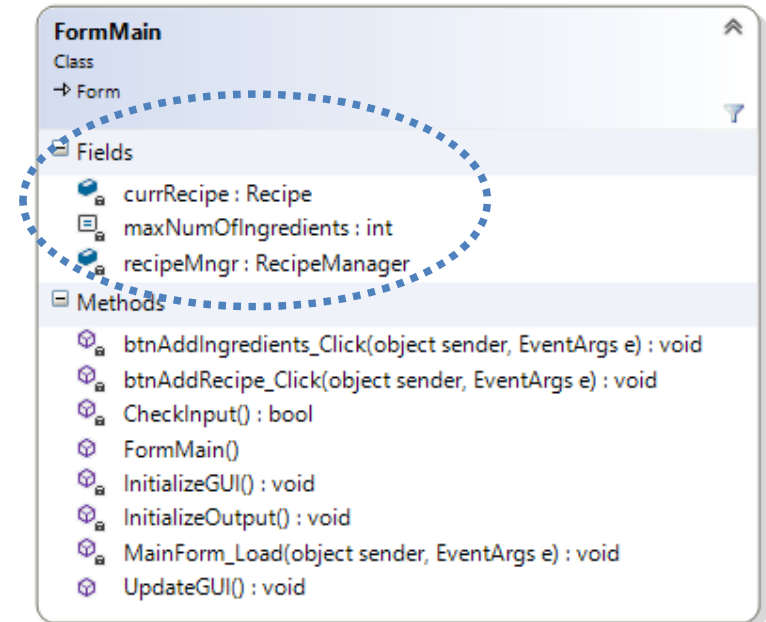
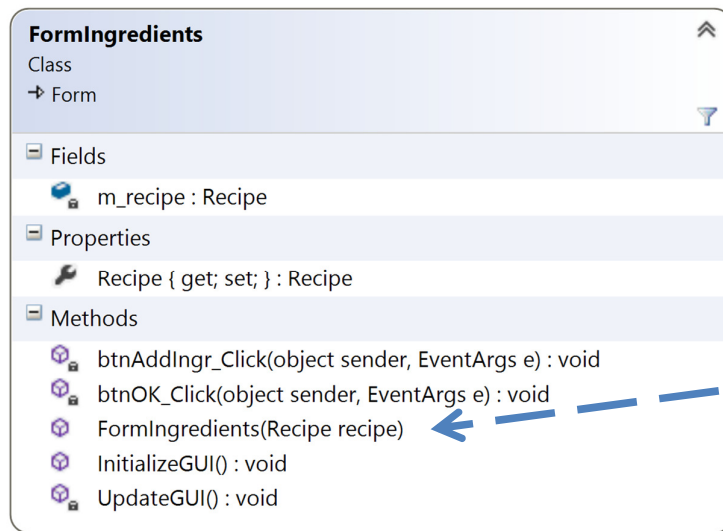
FoodCategory
 Enum

Meat
 Vegetarian
 Fish
 SeaFood

8.2 **currRecipe** is used to store temporarily the data for a recipe being inputted by the user. When all data is ready, the object is sent to the **recipeMngr** to be added in the array.

Important: Recreate currRecipe (currRecipe = new Recipe...) when you are done with one recipe and will start working with a new recipe. Why? A good question to discuss in the forum.

Note: Using the currRecipe makes life easier but it is meant that you must follow this solution. Take it as a hint!



Constructor with a parameter of the type Recipe. (You can change the constructor that is prepared by VS)·

FormMain passes the currRecipe to this form to fill with ingredients. An alternative to this is to use the set-property connected to m_recipe. (m_recipe is just a variable name; you may name it only "recipe")·

- 8.3 When the button **Add ingredients** is pressed, the **FormIngredients** can be loaded and shown as follows:

In the above code, the data is saved in the currRecipe but not yet added to **mngrRecipe** object. This is done when the user clicks the **Add recipe** button.

```
private void btnAddIngredients_Click(object sender, EventArgs e)
{
    FormIngredients dlg = new FormIngredients(currRecipe);
    DialogResult dlgResult = dlg.ShowDialog();

    if (dlgResult == DialogResult.OK)
    {
        if (currRecipe.CurrentNumOfIngredients <= 0)
        {
            MessageBox.Show("No ingriedients specified!");
            recipeMngr.Add(currRecipe);
            UpdateGUI();
        }
    }
}
```

8.4 Creating arrays and array elements:

- Every array must be created using the **new** keyword.
 - `private double [] priceList; //array declared but not created`
`//In the statement below, the array is created for storing maxItems double values`
`//The values can be directly saved as elements in the array.`
 - `priceList = new double[maxItems]; //array with elements of value type`
`//In the statement below, the array is created, but each element must also be`
`created before it can be saved in a position in the array.`
 - `loanList = new BankLoan[maxItems]; //array of objects`

- If you are using an array of value types (int, double, bool, etc) and string (which is not a value type), you don't need to create each element of the array before saving in the array. Array of ingredients is an array of strings. You only need to create the array, not each element.
- If you have an array of reference types, i.e. objects, you create the array first and then create each element of the array. For instance, if you have array of Recipes (as in the **RecipeManager**), to add a **Recipe** object in the array or save one at any position, the **Recipe** object must have been created using the keyword **new**, in addition to that the array itself is created:
 - Create the array: `private Recipe[] recipeArray = new Recipe[maxItems];`
 - `recipeArray[index] = new Recipe(maxIngredients);`
- An array that is created with a number of elements, cannot shrink or expand. To “delete” an element, you simply mark the element with zero-values for value types or **null** for reference type elements.
 - `pricelist [index] = -1.0;` //assuming that pricelist has elements of a double type.
 - `ingredientList[index] = null;`

8.5 Two things to differentiate when working with an array:

1. The capacity of an array, the number that you use to create an array. To depict this value after the array is created, you use the Length property of the array. **priceList.Length** gives the number of elements the array is created with.
2. The number of items currently saved. This is the number of elements in the array that have some valid data. This can be depicted by either

using a variable (numOfItems) that should be incremented every time you add a new value, decremented each time you rinse a value, or

by writing a method that returns the number of items with valid values. Use a loop through the array and count the number of elements that has a valid value (en element that is not vacant).

9. Help and Submission

A more detailed description of the classes are provided in a separate document for those who need more help and guidance.

Compress all your files and folders (particularly the folder Properties) that are part of your project into a ZIP or RAR file. Upload the compressed file via the same page where you downloaded the assignment.

Good Luck!

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

Farid Naisan,

Course Responsible and Instructor