

Apu Animal Park – Version 2

1 Objectives

Now that you are well trained in the first two pillars of OOP, encapsulation and inheritance, this assignment will give you training in polymorphism, using interface, and abstract classes. In the previous assignment, we also worked with dynamic binding which is a part of polymorphism. Through interfaces, we specify methods that a group of classes must implement without worrying about how the implementation is carried out by each class. Abstract classes are used to delegate some of the work to sub-classes. These two concepts together are key to implementation of polymorphism, and they have been used widely in well-known design patterns.

Polymorphism is what has made object-orientation powerful and extremely popular. Interfaces and abstract classes play a dominating role in designing software solution models, and they solve many recurring programming problems. It is very essential for every programmer to build skills in and use the power of polymorphism using object-orientation.

Another major improvement is using a collection to register an unlimited number of animal objects. We will write a class that manages the register, e.g. adding a new animal, changing existing data, removing an object and providing information on all the registered animal objects.

Notes:

- By “implement” it is meant to write code. Implementing a method implies to complete the method with a body.
- This assignment will be used again in the next module, so make sure to do a good job and keep your files for reuse. Do not forget to comment your code well and save your work quite often while you are writing code.
- This assignment is intended to be done using Windows Forms and controls. However, you may try Windows Presentation Foundation (WPF, which will examine in a later module) if you are an experienced Windows Forms programmer.
- You do not have to follow the instructions given here step by step and you may change the GUI, or completely redesign and implement your own solution, provided that the solution meets the minimum requirements, keep a good programming level, and follow the OOP rules well.

Copy your solution from the previous assignment to a new directory on your hard disk. Change the solution and project file names to match the name of the new assignment (e.g. Assignment2, or AnimalPartV2). This way, you have a good release version of your previous work

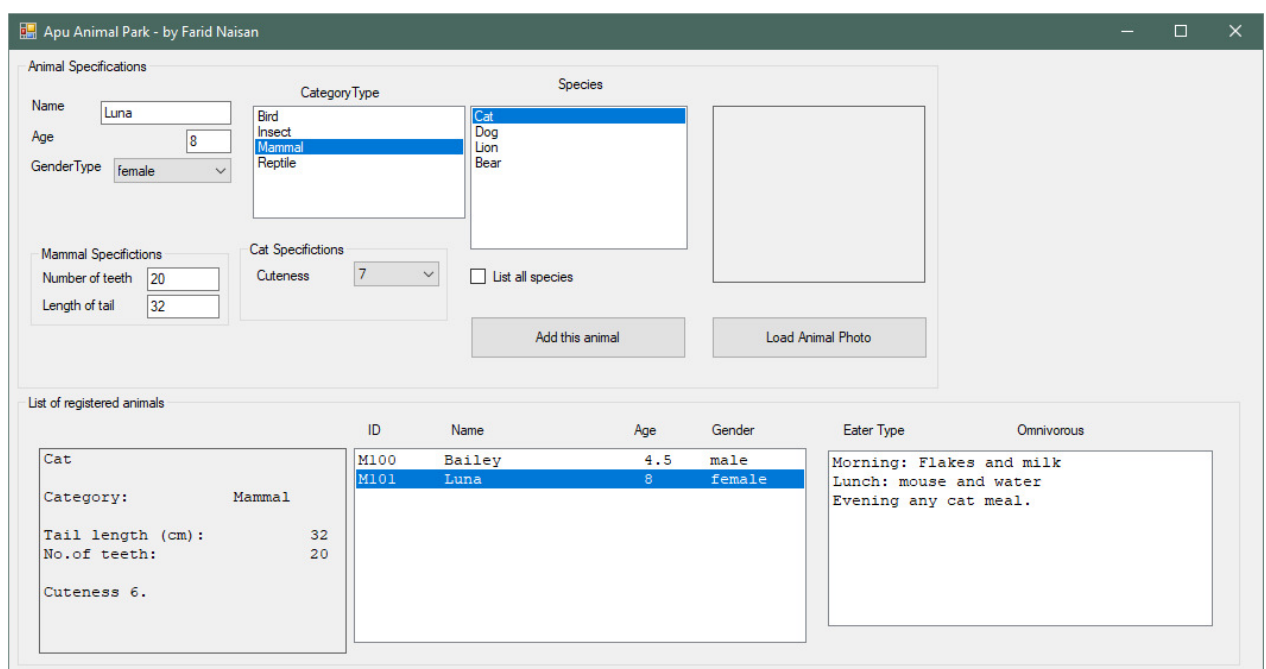
2 Description

In the previous assignment, you worked with a first version of the Animal Park project. Now, we are going to make a version 2 of the same project with some additional requirement specifications as outlined below.

New Features:

- 2.1 **IAnimal:** We are going to write an interface, **IAnimal** and make the classes **Animal**, **Mammal**, **Bird**, **Insect** and **Reptiles** as **abstract** classes. The interface helps us to make rules for the subclasses.
- 2.2 The **Animal** class has to implement **IAnimal** and provide implementation (i.e. code) to methods of **IAnimal**. By defining these classes as abstract, we are preventing instantiation of the classes, and this makes sense. The classes do not have complete information about an animal object, and thus are incomplete classes. They are, on the other hand, good base classes.
- 2.3 We are also going to practice with abstract and virtual methods. Define at least one of each type in the Animal class, i.e. one abstract and one virtual method.
- 2.4 **FoodSchedule:** Every species needs to have a feeding schedule, so the workers have instructions for preparing food and following a daily schema. We are going to write a class with a list of strings where every string contains some sort of info (an example is given later).
- 2.5 **EaterType:** Animal species (dog, butterfly, dove, etc.) can be classified according to whether they are a **Carnivore** (meat eater), **Herbivore** (plant eater) or **Omnivorous** (all eater) type. We are going to store this information within the **FoodSchedule** class in this version.
- 2.6 As mentioned above, every species should provide a food schedule consisting of a feeding plan for a particular species. To simplify the problem, you can hard-code the information in the species' classes (see example later).
- 2.7 **AnimalManager:** We are going to store animal objects in a collection of the type `List<Animal>`, and for this purpose, we are going to create a container class **AnimalManager**.

The figure below is a run-time example showing the new version in action:



The screenshot shows a Windows application titled "Apu Animal Park - by Farid Naisan". The interface is divided into several sections:

- Animal Specifications:**
 - Name:** Luna
 - Age:** 8
 - GenderType:** female
 - CategoryType:** A list box containing Bird, Insect, Mammal (selected), and Reptile.
 - Species:** A list box containing Cat (selected), Dog, Lion, and Bear.
 - Mammal Specifications:**
 - Number of teeth:** 20
 - Length of tail:** 32
 - Cat Specifications:**
 - Cuteness:** 7
 - Buttons:** "Add this animal" and "Load Animal Photo".
- List of registered animals:** A table with columns: ID, Name, Age, Gender, Eater Type, and Omnivorous.

	ID	Name	Age	Gender	Eater Type	Omnivorous
Cat	M100	Bailey	4.5	male	Morning: Flakes and milk	
	M101	Luna	8	female	Lunch: mouse and water	
					Evening: any cat meal.	

The Label component at the lower-left part of the figure, shows more details about the selected object. The information comes from the category class (Mammal) and the concrete class (Cat). The ListBox at the lower-right part displays a list of the items taken from the **FoodSchedule** object for the selected animal (Luna, Cat). (Note: for grade A, you are to use a **ListView** as described later).

3 Minimum requirements for all grades

The IAnimal interface

3.1 Create an interface **IAnimal** and define the following methods:

- A property (get and set) for name of all animals

```
string Name { get; set; }
```

- A property (get and set) for ID of all animals
- A property (get and set) for Gender of all animals
- A method returning an object of **FoodSchedule**

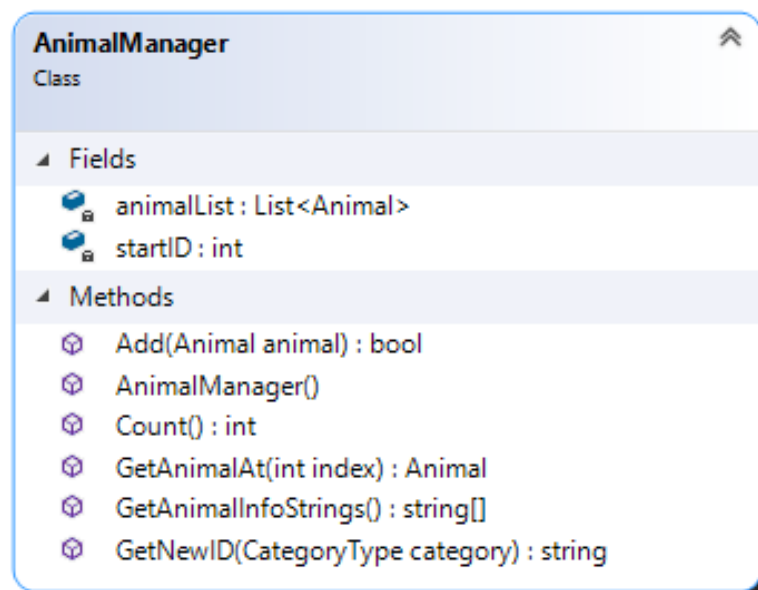
```
FoodSchedule GetFoodSchedule();
```

- A method returning a string made of an animal's specific information, e.g. information taken from the category of the animal object combined with data from the animal's object.

```
string GetExtraInfo();
```

The AnimalManager class

3.2 Create a new class **AnimalManager**. This class should maintain and handle a list of **Animal** objects. The class diagram below shows the structure of the class:



3.3 The instance variable **animalList** is a collection of the type `List<T>` where T is replaced by **Animal** (class).

- 3.4 The **Add** method is called by the **MainForm**, and it adds a new **Animal** object. Perform some validation of the object before adding, for instance, controlling so the object is not **null**.
- 3.5 You can set an ID to every animal object being added to the list. Referring to the GUI figure above, the IDs of the animals are set by the method **GetNewID** that combines a letter, e.g. **'M'** for Mammal, and the recent value of the **startID**. This is only an example. You can use any other simple method (like numbers 1, 2, ...) or a more complex system like GUID.
- 3.6 The method **GetAnimalInfoStrings** returns an array (string[]) by calling the **ToString()** method of every element of the list.

The Animal class and other abstract classes

- 3.7 Define the **Animal** class as **abstract** implementing **IAnimal**.

```
abstract class Animal: IAnimal
{
}
```

- 3.8 Define also the **Mammal**, **Bird** and other category-level classes with the modifier **abstract**.

Note: if you define a class without an access modifier (**public**, **internal**), the class will be available for all classes within the same namespace (default behavior **internal**). You can, however, use **public** with all classes and enum:s, as these are type definitions and often need to be available to other types. What is important is to be consequent using either pattern continuously.

- 3.9 Add the following code to the **Animal** class:

```
/// <summary>
/// The implementation of this method is to be provided by the subclasses.
/// The information is not available and therefore the job is delegated to subclasses
/// </summary>
/// <returns>An object of the FoodSchedule assigned to the particular object.</returns>
10 references
public abstract FoodSchedule GetFoodSchedule();

/// <summary>
/// prepare a string made of the specifications that are not included in the Animal
/// class.
/// A default implementation is coded in this method
/// </summary>
/// <returns></returns>
20 references
public virtual string GetExtraInfo()
{
    string strOut = string.Empty;

    strOut = string.Format("{0,-15} {1, 10}\n", "Category:", category.ToString());

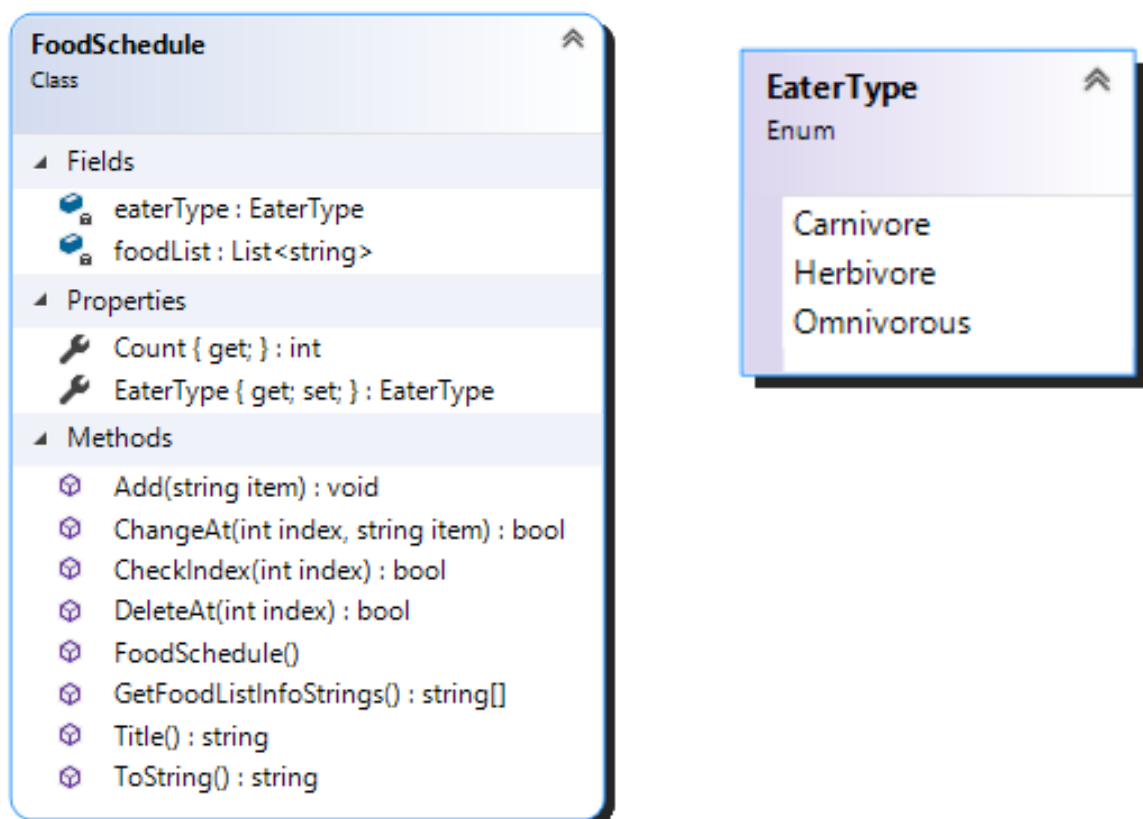
    return strOut;
}
```

The virtual method **GetExtrInfo** is intended to contain a string with values from the category-level classes (Mammals, etc.). For example, a Dog class must give information about the instance variables defined in the Dog class (e.g. Breed, Dog class) as well as data in the Mammal class. (numOfTeeth, tailLength). This method can be overridden in a concrete class such as the Dog class.

The abstract method **GetFoodSchedule()** needs not be implemented in the base classes (Mammal, etc.), but it must be implemented in the sub classes thereof, e.g. Dog, Cat, Falcon, etc.

The FoodSchedule class

3.10 Create a new class **FoodSchedule** and a new enum **EaterType**. The **FoodSchedule** class is intended to serve as schedule for feeding of the animals. In this version of the program, it is sufficient to construct a few necessary fields and methods, specified in the class diagram below.



3.11 The **CheckIndex** is a method that returns **false** if the given index is not within the range of the collection; otherwise, it returns true.

3.12 **GetFoodListInfoStrings** is a method that returns **foodList** as an array of strings (you may use the `ToArray()` method of the **List** collection). This array will be used to display the food schedule for a selected animal.

3.13 Override the abstract method **GetFoodSchedule** in the concrete classes. Go to the concrete classes (such as Dog, Falcon, etc.) that you have included in your project. Write code for this method here is a little hint (Dog class):

- 3.14 The instance variable **foodList** is a collection of string elements where each element is a line of text that describes a part of the feeding schedule. You may use an array instead. The strings of this list are displayed on the GUI, as can be seen on the figure given earlier. As an example, for a dog, the class Dog may provide a schema for a whole day, as exemplified by the image below.

```
private FoodSchedule foodSchedule;

//this method is called from the constructor
private void SetFoodSchedule()
{
    foodSchedule = new FoodSchedule();
    foodSchedule.EaterType = EaterType.Omnivorous;
    foodSchedule.Add("Morning: Flakes and milk");
    foodSchedule.Add("Lunch: bones and flakes");
    foodSchedule.Add("Evening any meat dish.");
}

public override FoodSchedule GetFoodSchedule()
{
    return foodSchedule;
}
```

Note: *The following part is mandatory only for grades A and B. If you are aiming at a pass grade C, you don't have to do the following part. Go directly to the submission section.*

4 Specifications and Requirements for a higher grade (A, B)

In addition to the above requirements, the following is also to be implemented:

Note: use of SortedList is not allowed.

For a grade B:

- 4.1 The **AnimalManager** class should have methods to sort the animal list by (1) name of animal and (2) by the specie type. The GUI should allow the user to select either of the sorting options. You can use any suitable type of components such as radio buttons or a ListView control with header-click capability. The sort order can be ascending or descending. It is enough to apply either one; make the choice by yourself in the code.

Hint 1: the **Animal** class should implement the interface **IComparable** (in addition to **IAnimal**) to allow sorting of animals after their names, or species (dog, cat). This way, you create one sorting possibility (default). To be able to sort in several ways you need to create a sorting method by yourself, implementing **IComparer<Animal>** and using it for the second sorting. You can follow the example given on the following webpage:

<https://learn.microsoft.com/en-us/troubleshoot/developer/visualstudio/csharp/language-compilers/use-icomparable-icomparer>

Hint 2: (Recommended)

Another hint is to skip implementing **IComparable** in **Animal** class, and instead, create two sorting classes like in the link below...In this way you only need one sorting method in the manager class, making a smarter use of interface as parameter.

<https://learn.microsoft.com/en-us/dotnet/standard/collections/comparisons-and-sorts-within-collections>

For a grade A:

- 4.2 Use a **ListView** for displaying a list of animals. However, if you solve this assignment as a MAUI-based application, you automatically fulfill the requirement for this part, provided you apply a sorting algorithm (the B part).

5 Grading and submission

Make sure that you submit the correct version of your project and that you have compiled and tested your project before handing in. Be careful not to use any hard-coded file paths (for example path to an image file on your C-drive) in your source code. It will not work on other computers. Projects that do not compile and run correctly will be directly returned for completion and resubmission.

Compress all the files, folders and subfolders into a **zip,7z** or **rar** file, and then upload it via the Assignment page in Canvas. Do not send your project via mail!

Good Luck!

Farid Naisan,

Course Responsible and Instructor