Pentalog

Module 01 – Week 04

Classes & Best Practices

DATA ANALYSIS
UX/UI
FRONT-END
TECHNOLOGY
NEAR/OFFSHORE
AGILITY
BACK-END
SPRINT
KANBAN
CAMPAIGNS
GROWTH HACKING
SCRUM
BACKLOG
DEVOPS
DESIGN
SEO
CONTINUOUS INTEGRATION
MOBILE
QA
AUTOMATION
RESPONSIVE
UNIT TESTING

Today's Agenda

Classes (part 1 of 2)

- Classes vs instances/objects
- Constructors
- Access modifiers
- Different types of assemblies
- Inheritance
- Destructors
- Fields
- Properties

Best practices for writing classes in C#

Our Code – Git Repo

https://github.com/nadiacomanici/PentastagiuDotNet2019Brasov

Classes

OOP = Object Oriented Programming

What is a Class?

- A class **groups variables (state) and methods (behavior)** and represents a **template** for creating a certain type of objects
  - The **instance** (**object**) of a class is a specific object created using the class template
- Each class should do a single thing (Single Responsibility Principle), so we need more classes to build an application.

```
// the class
public class Person
{
    // behavior and state
}

// the instance (or object)
Person nadia = new Person();
Person ovidiu = new Person();
```

Classes vs Objects/Instances

A CLASS

- The gingerbread cutter is the pattern for a gingerbread man.
- It defines the common elements for all gingerbread men:
  - 1 head, 1 mouth, 2 eyes
  - 2 hands, 2 feet, 2 buttons
- But you can’t eat it

OBJECTS (INSTANCES) OF A CLASS

- Each gingerbread man is created using that cutter and all gingerbread men have the same elements defined by the pattern:
  - 1 head, 1 mouth, 2 eyes
  - 2 hands, 2 feet, 2 buttons
- They can have some different properties
  - Coloring
  - Taste

Names for classes in C#

- The names should be nouns (singular) and UpperCamelCase

Good

```
public class SolarSystemControl
public class BoolToVisibilityConverter : IValueConverter
public partial class MainWindow
public class Student
```

Evil

```
public class UserControl1
public class MyConverter : IValueConverter
public partial class mainWindow
public class Students
```

Full Qualified Name

- If there are multiple classes with the same name, prefix with namespace to specify full qualified name of a class (so we an differentiate between them)

```
System.Windows.Shapes.Path pathShape;
System.IO.Path filePath;
```

Application requirements

- Each class contains:
  - **State** (nouns -> fields and properties)
  - Behavior (verbs -> methods or boolean properties)

- Requirements:
  - A university has two types of persons: students and teachers.
  - Each person has a first name, a last name and a birthdate.
  - Each student has an identifier and some marks
  - The application should display for each student the average of the marks, if he has a scholarship , if he is legally an adult and if he can vote.
  - The application should allow to sort the students by last name or average mark
  - In a similar manner, each teacher has a scientific title and can publish research papers

Identifying verbs and nouns

- Each class contains:
  - **State** (nouns -> fields and properties)
  - Behavior (verbs -> methods or boolean properties)

- Requirements:
  - A **university** has two types of **persons**: **students** and **teachers**.
  - Each **person** has a **first name**, a **last name** and a **birthdate**.
  - Each **student** has an **identifier** and some marks
  - The **application** should display for each **student** the **average** of the **marks**, it he has a **scholarship**, if he is legally an **adult** and if he can vote.
  - The **application** should allow to sort the **students** by **last name** or **average mark**
  - In a similar manner, each **teacher** has a **scientific title** and can publish **research papers**

Classes after text analysis

- University
  - **Students**
  - **Teachers**
  - *AddStudent()
  - *AddTeacher()

- Person
  - **FirstName**
  - **LastName**
  - **BirthDate**
  - ***Age**

- Application
  - DisplayStudents()
  - SortStudentsByLastName()
  - SortStudentsByAverageMark()

- Student is a Person
  - **Id**
  - **Marks**
  - **AverageMark**
  - *HasScolarship*
  - *IsLegallyAdult*
  - *CanVote*
  - ***FirstName**
  - ***LastName**
  - ***BirthDate**

- Teacher is a Person
  - **ScientificTitle**
  - **ResearchPapers**
  - ***FirstName**
  - ***LastName**
  - ***BirthDate**

Creating Instances

1. Declaration	- The **intention** - We say what is the type (the class) of the object and give it a name - The object is allocated a place in memory (address) where it will be placed after creation. But now that place doesn't contain the object (is not initialized) - We cannot access members of an object if that object is not initialized/created	- „I am going to make a ginger bread man for Tom and put it in the second shelf of the fridge" - At this step, if Tom comes and opens the fridge and looks at the second shelf, he will not see the gingerbread man and he cannot take or eat it => exception
2. Initialization (Creation)	- The **action** - Actually creating the object in memory - We create a valid object that can be found in that place in memory	- „I create the gingerbread man and placed it on the second shelf of the fridge"

Creating Instances

- To create an object/instances we need 2 steps:
  - Declaration
  - Initialization / Creation
- These 2 steps can be made separately or in a single instruction:

```
// step 1: declaration
// at this point, 'nadia' object is null (not initialized)
Person nadia;

// step 2: initialization
nadia = new Person();
```

```
// step 1+2: declaration and initialization
Person nadia = new Person();
```

- You cannot access members of an object if that object is not initialized.

Constructors

- Each time an object is initialized/created, its constructor is getting called.
- Using constructors, we create instances of a class
- The constructor should create a valid (operational) instance of a class
- A class can have multiple constructors, but they need to have different parameter lists (constructor signature)
- Types of constructors:
  - Default constructor (auto generated at runtime)
  - Constructor without parameters (defined by the developer)
  - Constructor with parameters (defined by the developer)
  - Static constructor (defined by the developer)

Default Constructor

```
public class Person
{
    public string FirstName;

    // the class doesn't have any defined constructor, so at runtime
    // a default constructor without parameters will be generated
}
```

```
static void Main(string[] args)
{
    // the object is not initialized, so there will be a compile error
    // 'Use of unassigned variable'
    Person person;
    person.FirstName = "John";

    // correctly initialized and used
    Person anotherPerson = new Person();
    Console.WriteLine(anotherPerson.FirstName);
    anotherPerson.FirstName = "Ionel";
    Console.WriteLine(anotherPerson.FirstName);
}
```

Constructor without parameters

- Inside the constructor without parameters, usually members are created and assigned default values so that the object is valid
- If the developer defines a constructor (with or without parameters) inside a class, the default constructor will no longer be auto generated

```
public class Person
{
    public string FirstName;
    public string LastName;

    public Person()
    {
        this.FirstName = "John";
        this.LastName = "Doe";
    }
}

Person person = new Person();
Console.WriteLine(person.FirstName);
Console.WriteLine(person.LastName);
```

Constructor with parameters

- Inside the constructor with parameters, usually members are created and assigned values associated with the parameters, so that the object is valid
- If the developer defines a constructor (with or without parameters) inside a class, the default constructor will no longer be auto generated

```
public class Person
{
    public string FirstName;
    public string LastName;

    public Person(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}

// compile error because
// the default constructor is no longer generated
Person person = new Person();

// correct
Person anotherPerson = new Person("Ionel", "Popescu");
```

Multiple Constructors

- A class can have multiple constructors, but they need to have different parameters
- This way, an object can be created in multiple ways
- Constructor overloading

```
public class Person
{
    public string FirstName;
    public string LastName;

    public Person()
    {
        this.FirstName = "John";
        this.LastName = "Doe";
    }

    public Person(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}

Person person = new Person();
Person anotherPerson = new Person("Ionel", "Popescu");
```

Constructor with default parameters

- Each parameter can have a default value which will be used in case the developer doesn't specify one
- Allows constructor overloading
- The default parameters must be at the end of the parameter list in the constructor's definition

```
public class Person
{
    public string FirstName;
    public string LastName;

    public Person(string firstName = "John", string lastName = "Doe")
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}

Person person = new Person();
Console.WriteLine($"{person.FirstName} {person.LastName}");

Person secondPerson = new Person("Ionel");
Console.WriteLine($"{secondPerson.FirstName}
{secondPerson.LastName}");

Person thirdPerson = new Person("Ionel", "Popescu");
Console.WriteLine($"{thirdPerson.FirstName} {thirdPerson.LastName}");
```

Calling a constructor from another constructor

- Use **this** to refer to the current object

```
public class Person
{
    public string FirstName;
    public string LastName;

    public Person() : this("John", "Doe")
    {
    }

    public Person(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}
```

Access Modifiers

Or: Visibility Modifiers / Specifiers

Assemblies

- Different types of projects:
  - **Class Library** -> output is UniversityLibrary.dll
  - **Console Application** -> output is UniversityConsoleApp.exe

- Add reference between projects:
  - In the console app, add a reference to the class library and use the functionality implemented in the class library
  - Advantages:
    - Separate the implementation from the user interface (UI)
    - This way, we can use the same functionality with different types of UI
    - The class library should be UI independent

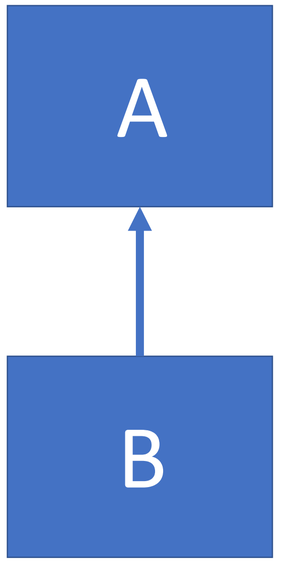
Access Modifiers

- **Public** – visible from everywhere, there are no restrictions
- **Internal** – visible inside the assembly, but not outside it
- **Private** – visible only in the class it is defined
- **Protected** – visible in the class it is defined and all the classes that derive from it
- Defaults:
  - Each class is internal by default
  - Each member of a class is private by default
  - Each member of a struct is private by default
  - Each member of a interface or an enum is public by default

Inheritance

Inheritance

- If there's a relationship between classes „B is a A", then we have inheritance between the classes and we say:
  - A is the parent class (base class)
  - B is the child class (derived class)
- All public, internal and protected members of class A belong to class B as well and can be used inside class B
- All private members of class A can be used only inside class A



Inheritance - Example

```
// class must be public to be seen from a different project
public class Person
{
    public string FirstName;
    protected string lastName;
    private DateTime birthDate;
}

public class Student : Person
{
    public Student(string firstName, string lastName, DateTime dateOfBirth)
    {
        // correct, because it is public
        this.FirstName = firstName;

        // correct, because it is protected
        this.lastName = lastName;

        // error at compile time, because it is private
        this.dateOfBirth = dateOfBirth;
    }
}
```

Calling a base constructor from a derived class

- Use **base** to refer to the parent object

```
public class Person
{
    public string FirstName;
    public string LastName;

    public Person(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}

public class Student : Person
{
    public Student(string firstName, string lastName)
                : base(firstName, lastName)
    {
    }
}
```

Static Constructor

- The static constructor is used only for initializing static members of a class
- The static constructor cannot have an access modifier
- The static constructor is called once, not each time a new object is created

```
public class Person
{
    public static int MinimumAgeForVoting;
    public string FirstName;

    static Person()
    {
        // correct call
        MinimumAgeForVoting = 18;

        // incorrect call, since only static members can be accessed
        // FirstName = "John";
    }
}
```

Destructor

Destructors

```
public class Student
{
    // destructor
    ~Person()
    {
    }
}
```

- Only classes can have destructors
- A class can have only one destructor (maximum 1)
- Destructors are not called by developers, but are automatically called by the Garbage Collector when it frees the memory for that object
- Destructors cannot have parameters or access modifiers
- The destructor doesn’t have to be defined, only if there are special resources that need to be freed once the object no longer exists
- More: http://msdn.microsoft.com/en-us/library/66x5fx1b.aspx

Fields & Properties

Fields

- Fields represent any type of variables inside a class
- You should make the fields private or protected and make a getter/setter for it if it needs to be visible from outside the class
- Private and protected fields are lowerCamelCase

Good

```
private int nextIndex;
protected List<Student> students;
```

Fields

- Default values for fields (if you don’t initialize them)
  - If the field is a value type instance, then it will be automatically assigned the default value of that type
  - If the field is a reference type instance, then it will be null

```
public enum Gender
{
    Male, Female
}

public class NoteBook
{
}

public class Person
{
    // default value is null
    public string FirstName;
    public string LastName;

    // default value is 1/1/0001 12:00:00 AM
    protected DateTime birthDate;

    // default value is first enum value
    protected Gender gender;
}

public class Student : Person
{
    // default value is 0
    private int Id;

    // default value is null
    private NoteBook notebook;
}
```

Static fields

- The static fields belong to the class, not to the instances
  - Static fields are shared between all instances of a class
- You don’t need an instance to access a static field
- If you want to use them from outside the class, you should prefix the static field name with the class name

```
public class Person
{
    public static int MinimumAgeForVoting = 18;

    //....
}

Console.WriteLine(Person.MinimumAgeForVoting);
```

Constant fields

- A constant is a variable that has a known value at compile time and doesn't change its value during runtime.
- The constant fields belong to the class, not to the instances
  - constant fields are shared between all instances of a class
  - there is no need to use the "static" keyword when defining them
- Constants are used similar to static fields, by prefixing with the class name

```
public class Person
{
    public const int MinimumAgeForId = 14;

    //....
}

Console.WriteLine(Person.MinimumAgeForId);
```

Readonly fields

- A readonly field is a variable that doesn't have a known value at compile time. The value is set at runtime and doesn't change its value during the lifetime of the application.
  - You should assign it when you define it or in the constructor
- The readonly fields belong to the instances, not to the class
  - You need a instance to access it and can have different values for each instance

```
public class Person
{
    public readonly int MinimumAgeForRetirement;
    // ...
    private Gender gender;

    public Person(string firstName, string lastName, Gender gender)
    {
        this.MinimumAgeForRetirement = gender == Gender.Male ? 65 : 63;
    }
}
```

Fields – Best Practices

- This is not a best practice, to leave the field public
- Fields should be **private/protected**

```
public class Person
    {
        // this is not a best practice
        // to leave the field public
        public string FirstName;
}
```

Naming Fields

- Each property and field should have a good name, that represents its role
- According to the access modifier, the casing should be:
  - public/internal -> UpperCamelCase
  - private/protected -> lowerCamelCase

Good

```
public class StudentList
{
    private List<Student> students;
    private int nextId;
}
```

Evil

```
public class StudentList
{
    private List<Student> l;
    private int _next_Id;
}
```

Properties

- A property is a member that allows:
  - accessing/modifying a field
  - or computing a value
- A property must have at least a get or a set, each with different access modifiers
- Properties are called similar to fields (without parenthesis, like methods)

```
public class Person
{
    private string firstName;
    public string FirstName
    {
        get
        {
            return firstName;
        }
    }

    public string LastName { get; set; }
    public DateTime BirthDate { get; protected set; }
    public Gender Gender { get; protected set; }
}
```

Properties – the old way

- The C++ way is to create getters/setters as methods, to access/modify the field

```
public class Person
{
    private string firstName;

    public string GetFirstName()
    {
        return this.firstName;
    }

    internal void SetFirstName(string newFirstName)
    {
        this.firstName = newFirstName;
    }
}
```

Properties – the C# way

- The C# way is to create getters/setters as properties

```
public class Person
{
    private string firstName;

    public string FirstName
    {
        get { return this.firstName; }
        private set { this.firstName = value; }
    }
}
```

- Or even more compact (syntactic sugar)

```
public class Person
{
    public string FirstName { get; private set; }
}
```

Properties – setter validations

- Inside the setter, you can make additional validations before setting the value to the inner field
- The new value is stored into the value variable

```
public class Person
{
    private string firstName;
    public string FirstName
    {
        get
        {
            return firstName;
        }
        set
        {
            if (string.IsNullOrEmpty(value) == false)
            {
                firstName = value;
            }
        }
    }
}
```

Computed Properties

- A property can have only a getter and compute a value using some internal data for that object

```
public class Person
{
    public const int MinimumAgeForId = 14;
    public static int MinimumAgeForVoting = 18;

    public string FirstName { get; private set; }
    public string LastName { get; private set; }
    public DateTime BirthDate { get; private set; }
    public double Age
    {
        get
        {
            return (DateTime.Now - this.BirthDate).TotalDays / 365.2425;
        }
    }

    public bool CanVote
    {
        get
        {
            return this.Age > MinimumAgeForVoting;
        }
    }
}
```

Naming Properties

- Each property and field should have a good name, that represents its role
- Getters and setters should be adapted to the standard of the language
- Use “Is/Are/Can/Has” for a property that returns a boolean value

```
public bool CanVote
{
    get
    {
        return Age > MinimumAgeForVoting;
    }
}

public bool HasIdCard()
{
    return this.Age > MinimumAgeForId;
}

public bool IsLegallyAdult
{
    get
    {
        return Age > MinimumAgeForVoting;
    }
}
```

Properties – best practices

- When creating a field, make it private and change it to something more visible only when needed
- When you create a property, make the setter private and change it to something more visible only when needed

Constant Fields instead of Magic Numbers (1)

- Magic numbers are hardcoded values whose value are not obvious for someone looking the first time at the code
- They can generate bugs because:
  - Someone that doesn't know what the value represents, can change it incorrectly
  - If it is used in multiple places and the dev forgets to replace it in all the places, there will be bugs

Evil

```
return (DateTime.Now - DateOfBirth).TotalDays / 365.2425;
size = size + 729;
```

Constant Fields instead of Magic Numbers (2)

- If it is used in multiple places and the developer forgets to replace it in all the places, there will be bugs

Evil

```
public class Person
{
    public bool CanVote
    {
        get
        {
            return Age > 18;
        }
    }

    public bool IsLegallyAdult
    {
        get
        {
            return Age > 21; //18;
        }
    }
}
```

Good

```
public class Person
{
    public static int MinimumAgeForVoting = 18;

    public bool CanVote
    {
        get
        {
            return Age > MinimumAgeForVoting;
        }
    }

    public bool IsLegallyAdult
    {
        get
        {
            return Age > MinimumAgeForVoting;
        }
    }
}
```

What’s next?

- Next week, we’ll continue to:
  - implement methods and functionality
  - create the university and the list of students
  - create a service for retrieving the students

Homework

- Create the classes from the following requirements:
  - Create an application that allows users to post messages on a common board.
  - A person can create an account using his email and personal information like first name, last name, birthdate.
  - Each post should have an author
  - The board should display all the posts, created by all the users, chronologically, in descending order (latest first)
- Notes:
  - No implementation for methods, we will continue next week with that
  - Create a different repository on GitHub for this homework
  - Create 2 projects in the same solution: a class library and a console app