Pentalog

Module 01 – Week 05

Classes & Best Practices (part 2)

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 2 of 2)

- Methods
  - Parameters: out, ref, params, value and reference types, optional
  - Static methods
  - Extension methods
  - Override vs overload
- Partial classes
- Abstract and sealed classes

Further:

- Collections, Students Service and Implementing Methods for UniversityApp
- Interfaces
- Best practices for writing classes in C#

Our Code – Git Repo

https://github.com/nadiacomanici/PentastagiuDotNet2019Brasov

Classes - methods

OOP = Object Oriented Programming

Methods

- Methods represent the behaviour of a class and should contain verbs in their names
- Each method has an unique signature defined by the combination of name and parameters
- Each method has:
  - An access modifier
  - A return value
  - A name
  - (optional) Parameters

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

```
class Program
{
    static void Main(string[] args)
    {
        //...
    }
}
```

Methods – return type: void

- If a method doesn't return anything, the return type should be void
- You can use 'return' to get out of that method
- It's not mandatory to write a return statement

```
public void DisplayNumbers(List<int> numbers)
{
    foreach (int number in numbers)
    {
        Console.WriteLine(number);
    }
}

public void DisplayNumbersUntilZero(List<int> numbers)
{
    foreach (int number in numbers)
    {
        if (number == 0)
        {
            return;
        }
        Console.WriteLine(number);
    }
}
```

Methods – return objects

- If a method returns something, all paths inside that method should lead to a „return …” line
- The type of the returned object must match the return type of the method

```
public double ComputeAverage(List<int> numbers)
{
    if (numbers != null && numbers.Count > 0)
    {
        double sum = 0;
        foreach (int number in numbers)
        {
            sum += number;
        }
        return sum / numbers.Count;
    }
    // If we don't write this line, we will
    // get a compile error that:
    //not all code paths return a value
    return 0;
}
```

Static Methods

- If a method is static, then it can only use only static members of that class
- You cannot access ‘this’ inside a static method

```
// If we don't declare the method as static
// we will get a compile time error because
// Main method is static and can call only other
// static methods
public static void DisplayNumbers(int[] numbers)
{
    foreach (int number in numbers)
    {
        Console.WriteLine(number);
    }
}

static void Main(string[] args)
{
    // declare and assign collection
    int[] numbers = new int[] { 1, 2, 3, 4, 5, 6 };
    DisplayNumbers(numbers);

    // declare a fixed size collection and assign
values afterwards
    int[] integers = new int[3];
    integers[0] = 10;
    integers[1] = 11;
    integers[2] = 12;
    DisplayNumbers(integers);
}
```

Methods – best practices

- Methods represent the behaviour of a class and should contain verbs in their names
- Each method should do a single thing, as described in the method name
- When creating a method, set is as private and change the access modifier only when it is needed from outside the class
- All methods in C# (no matter the access modifier) should be UpperCamelCase
- All parameters of the methods in C# should be lowerCamelCase

Methods – best practices

Good

```
public class StudentList
{
    public int Count() {...}
    public void Add(Student student) {...}
    private void GetIndexOf(Student student) {...}
}
```

Evil

```
public class StudentList
{
    public int count() {...}
    public void Add1(Student student) {...}
    private void getIndexOf(Student student) {...}
}
```

Methods – passing arguments

- An argument is the concrete value sent for a parameter, when you call the method
- The parameters of a method can be:
  - a value type
  - a reference type
  - ref
  - out
  - in
  - params

Value Types vs Reference Types - parameters

```
static void IncrementNumber(int number)
{
    number++;
}

private static void ValueParametersSample()
{
    int x = 32;
    Console.WriteLine("Before: x = {0}", x);
    IncrementNumber(x);
    Console.WriteLine("After: x = {0}", x);
}
```

```
static void SetBookName(Book book)
{
    book.Name = "Alice in wonderland";
}

private static void ReferenceParametersSample()
{
    Book book = new Book();
    Console.WriteLine($"Before: {book.Name}");

    SetBookName(book);
    Console.WriteLine($"After: {book.Name}");
}
```

Value Types vs Reference Types

Value types	Reference types
When assigning X to variable and X is of type value – the content of X is **copied** from the place where X is in memory where the variable is in memory	When assigning Y to variable and Y is of type reference – the content of Y is only **referenced**, not copied. The variable will point to the same place in memory where Y is
Sending parameters – makes a copy	Sending parameters – sends a reference (pointer)
Default value: type default value	Default value: null
Cannot be null	Can be null
struct	class

Methods – ref parameters

- The „ref” keyword indicated that an argument is passed by reference
- Any change to this argument inside the method will remain after coming back from that method
- An argument sent to a ref parameter must be initialized
- The method and the call must use „ref” keyword
- More: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref

```
public void IncrementNumber(ref int x)
{
    x++;
}

int number = 3;
Console.WriteLine($"Before: {number}");
IncrementNumber(ref number);
Console.WriteLine($"After: {number}");
```

Methods – out parameters

- Similar to „ref”, but doesn’t require the parameter to be initialized before the call
- You must assign a value to the parameter inside the method
- Avoid using it, instead return the modified value
- The method and the call must use „out”
- More: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref

```
public void SetNumberToOne(out int x)
{
    x = 1;
}

int a;
// this line has a compile error because
// you cannot use an unassigned variable
// Console.WriteLine($"Before: {a}");
// but you can send it as an out parameter
SetNumberToOne(out a);
Console.WriteLine($"After: {a}");

int b = 4;
Console.WriteLine($"Before: {b}");
// you can also send an initialized
argument
// as an out parameter
SetNumberToOne(out b);
Console.WriteLine($"After: {b}");
```

Methods – params parameters (varadic)

- You can specify a method parameter that takes a variable number of arguments
- You can have a single params keyword in a method's declaration
- We already used params
  - String.Format(format, paramsArguments)
  - Console.WritelLine(format, paramsArguments)

```
public int ComputeSum(params int[] numbers)
{
    int sum = 0;
    foreach (int number in numbers)
    {
        sum += number;
    }
    return sum;
}

Console.WriteLine(ComputeSum());
Console.WriteLine(ComputeSum(1));
Console.WriteLine(ComputeSum(1, 12));
Console.WriteLine(ComputeSum(1, 2, 3, 4, 5, 6));
```

Methods – optional parameters

- The definition of a method can specify if a parameter is optional or not. In case it is optional, it can be omitted when the method is called
- Each parameter can have a default value which will be used in case the developer doesn't specify one:
  - A constant expression
  - An expression of the form new ValueType()
  - An expression of the form default(ValueType)
- The default parameters must be at the end of the parameter list in the method's definition
- Allows method overloading

```
public int MultiplyNumber(int x, int factor = 1)
{
    return x * factor;
}

Console.WriteLine(MultiplyNumber(3));
Console.WriteLine(MultiplyNumber(3, 1));
Console.WriteLine(MultiplyNumber(3, 2));
```

All instances are objects

- Every class derives from object (even if it is not explicitly declared)
- All objects will have:
  - ToString()
  - GetHashCode()
  - Equals(obj)

```
public class Book
{
    public string Name { get; private set; }
    public Book(string name)
    {
        this.Name = name;
    }

    // if we don't override the 'ToString()' method
    // the Console.WriteLine call will display
    // the full qualified name of the class
    // instead of something specific to the instance
    public override string ToString()
    {
        return this.Name;
    }
}
List<object> allKindsOfObjects = new List<object>();
allKindsOfObjects.Add(1);
allKindsOfObjects.Add("String here");
allKindsOfObjects.Add(new Book("Ion"));
foreach (object obj in allKindsOfObjects)
{
    Console.WriteLine(obj);
}
```

Override and overload

- Override (Ro: suprascriere) = different implementation of a method in a derived class (the same method signature)
  - Method 1: Virtual and override keywords
    - Use „virtual” keyword in base class to show that a method can be overridden in the derived classes
    - Use „override” keyword in the derived class
  - Method 2: New keyword
  - Properties and methods can be overridden
- Overloading (Ro: supraincarcare) - define the same method name with different parameters
  - Constructors and methods can be overloaded

Overriding methods and properties

```
public class Person
{
    public virtual string FullName
    {
        get
        {
            return $"{FirstName} {LastName}";
        }
    }

    // ToString() is virtual in object class
    // and every class in C# derives from object
    public override string ToString()
    {
        return FullName;
    }
}
```

```
public class Teacher : Person
{
    // default value is null
    public string ScientificTitle { get; private set; }

    public override string FullName
    {
        get
        {
            return $"{ScientificTitle} {FirstName} {LastName}";
        }
    }
}
```

Overloading constructors and method

```
public class Person
{
    public Person() : this("John", "Doe", DateTime.Now, Gender.Male)
    {

    }

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

Abstract classes

Abstract classes

- The abstract class allows you to create incomplete classes or class members that need to be implemented into a derived class
- The incomplete method doesn’t have a body and the method definition ends with „;”
- An abstact class cannot be instatiated
- The derived class:
  - either implements all the incomplete members and thus can be instantiated
  - either doesn’t implement all the incomplete members and is abstract as well
- When implementing in the derived class, you have to use the override keyword

Abstract classes

```
public abstract class MultiplicationBaseClass
{
    public abstract int MultiplyNumber(int x);
}

public class ThreeTimesMultiplication : MultiplicationBaseClass
{
    public override int MultiplyNumber(int x)
    {
        return 3 * x;
    }
}

public class NoMultiplication : MultiplicationBaseClass
{
    public override int MultiplyNumber(int x)
    {
        return x;
    }
}

public class TenTimesMultiplication : MultiplicationBaseClass
{
    public override int MultiplyNumber(int x)
    {
        return 10 * x;
    }
}
```

```
List<MultiplicationBaseClass> multiplications;
multiplications = new List<MultiplicationBaseClass>();

// compile time error because
// you cannot create an abstract class instance
// multiplications.Add(new MultiplicationBaseClass());
multiplications.Add(new ThreeTimesMultiplication());
multiplications.Add(new NoMultiplication());
multiplications.Add(new TenTimesMultiplication());

int x = 3;
foreach (MultiplicationBaseClass multiplication in
multiplications)
{
    Console.WriteLine(multiplication.MultiplyNumber(x));
}
```

Polymorphism

- The ability of an object to take “multiple shapes”
  - Objects of a derived type can be treated as objects of the base class
  - Virtual methods allow base classes to have their own definitions and implementations
- More: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/polymorphism

Sealed

Sealed Classes

- Sealed classes cannot be derived from

```
public sealed class SealedClass
{

}

// cannot derive from a sealed class
public class ImpossibleClass : SealedClass
{

}
```

Sealed Methods

- If you define a method to be sealed, then any derived class cannot override it
- The sealed class must be override as well (this means it is virtual in its base class)

```
public class BaseClass
{
    public virtual void DoSomething()
    {

    }
}

public class BaseClassWithSealedMethod : BaseClass
{
    // a sealed method must be override too
    public sealed override void DoSomething()
    {

    }
}
```

Partial classes

Partial classes

- Allows to define a class in multiple files
- Namespace and class name must match
- You must use “partial” before the “class” keyword

```
namespace PartialClassesSample
{
    public partial class Student
    {
        public string LastName { get; private set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Student student = new Student("Ion", "Vasile");
            Console.WriteLine($"{student.FirstName} {student.LastName}");
        }
    }
}
```

```
namespace PartialClassesSample
{
    // must specify partial before class
    // and namespace must match for both partial classes
    public partial class Student
    {
        public string FirstName { get; private set; }

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

Static classes

Static classes

- A static class contains only static methods
- A static class cannot be instantiated or derived from
- More: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members

Extension methods

Extension Methods

- Must be defined in a static class
- The method should be static and the first parameter prefixed with “this”
- In order to use it, add a using with its namespace
- More: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods

```
namespace ExtentionMethodsSample.Helpers
{
    public static class StringExtentionMethods
    {
        public static bool IsEmpty(this string str)
        {
            return (str == null
                || string.IsNullOrEmpty(str.Trim())
                || string.IsNullOrWhiteSpace(str.Trim()));
        }
    }
}
```

```
using System;
using ExtentionMethodsSample.Helpers;

namespace ExtentionMethodsSample
{
    class Program
    {
        static void Main(string[] args)
        {
            string str = "   ";
            Console.WriteLine(str.IsEmpty());
        }
    }
}
```

Extracting methods from text

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**

Collections

- Collections = lists of items
- Collections can be generic of a certain type or of type object
- Collections can be:
  - Fixed sized
  - Dynamic sized
- .Net offers a lot of types of collections with build-in functionality: Array, List<T>, Stack<T>, Dictionary<Key, Value>, HashTable etc.

```
// declare and assign collection
int[] numbers = new int[] { 1, 2, 3, 4, 5, 6 };
```

```
// declare a fixed size collection and assign values afterwards
int[] integers = new int[3];
integers[0] = 10;
integers[1] = 11;
integers[2] = 12;
// get the number of items in the list
int count = integers.Length;
```

```
// declare and create a dynamic-size collection
List<int> genericList = new List<int>();
// add as many items as you want
genericList.Add(1);
genericList.Add(13);
genericList.Add(21);
genericList.Add(33);
// get the number of items in the list
int genericListCount = genericList.Count;
// remove item at specified index
genericList.RemoveAt(1);
// remove specified item
genericList.Remove(21);
// remove all items
genericList.Clear();
```

Students Service

- We will replace the List<Students> with a StudentService inside the University class
- When working with a collection of objects, there are 4 basic operations that can be done (CRUD):
  - Create (add new items)
  - Read/Retrieve (get items)
  - Update (update item)
  - Delete (delete item)

Forwarding calls to StudentsService

```
public class University
{
    private StudentsService studentsService;

    public University()
    {
        this.studentsService = new StudentsService();
    }

    public IEnumerable<Student> GetStudents()
    {
        return studentsService.GetStudents();
    }

    public Student GetStudentById(int id)
    {
        return studentsService.GetStudentById(id);
    }

    public Student AddStudent(string firstName, string lastName, DateTime birthDate, Gender gender)
    {
        return studentsService.AddStudent(firstName, lastName, birthDate, gender);
    }

    public bool DeleteStudentById(int id)
    {
        return studentsService.DeleteStudentById(id);
    }

    public bool UpdateStudent(int id, string newFirstName, string newLastName)
    {
        return studentsService.UpdateStudent(id, newFirstName, newLastName);
    }
}
```

Students Service - Generating methods

```
internal class StudentsService
{
    // Create
    public Student AddStudent(string firstName, string lastName, Gender gender, DateTime birthDate)
    {
        throw new NotImplementedException();
    }

    // Retrieve
    public List<Student> GetStudents()
    {
        throw new NotImplementedException();
    }
    public Student GetStudentById(int id)
    {
        throw new NotImplementedException();
    }

    // Update
    public bool UpdateStudent(int id, string newFirstName, string newLastName)
    {
        throw new NotImplementedException();
    }

    // Delete
    public bool DeleteStudentById(int id)
    {
        throw new NotImplementedException();
    }
}
```

Students Service – Adding students

- We need to have a mecanism to generate new ids each time a new student is added

```
internal class StudentsService
{
    private List<Student> students;
    private static int nextId = 1;

    public StudentsService()
    {
        this.students = new List<Student>();
        AddStudent("Vasile", "Popescu", new DateTime(1990, 03, 02), Gender.Male);
        AddStudent("Maria", "Ionescu", new DateTime(1988, 02, 24), Gender.Female);
        AddStudent("Ionel", "Georgescu", new DateTime(1991, 11, 13), Gender.Male);
    }

    public Student AddStudent(string firstName, string lastName, DateTime birthDate, Gender gender)
    {
        Student student = new Student(nextId++, firstName, lastName, birthDate, gender);
        this.students.Add(student);
        return student;
    }
```

Students Service – Retrieving students

- You can get the entire list or one of the items in the list, by its id

```
public List<Student> GetStudents()
{
    return this.students;
}

public Student GetStudentById(int id)
{
    foreach (Student student in this.students)
    {
        if (student.Id == id)
        {
            return student;
        }
    }
    return null;
}
```

Using University in Console App

```
class Program
{
    private static void DisplayStudent(Student student)
    {
        Console.WriteLine("{0} {1} (id={2}) - {3} - {4}",
            student.FirstName, student.LastName, student.Id, student.Gender, student.BirthDate);
    }

    static void Main(string[] args)
    {
        University university = new University();
        university.AddStudent("Nadia", "Comanici", new DateTime(1986, 01, 24), Gender.Female);
        university.AddStudent("Radu", "Popescu", new DateTime(1990, 10, 13), Gender.Male);

        Console.WriteLine("All students:");
        foreach (Student student in university.GetStudents())
        {
            DisplayStudent(student);
        }
        Console.WriteLine();

        Console.Write("Search student by id: ");
        int id = int.Parse(Console.ReadLine());
        Student foundStudent = university.GetStudentById(id);
        if (foundStudent != null)
        {
            DisplayStudent(foundStudent);
        }
        else
        {
            Console.WriteLine("No student with that id");
        }
    }
}
```

Students Service – Deleting a student

- Delete by id - if student is found
- Return a boolean value to indicate the operation was successful

```
public bool DeleteStudentById(int id)
{
    Student student = GetStudentById(id);
    if (student != null)
    {
        students.Remove(student);
        return true;
    }
    return false;
}
```

Students Service – Updating a student

- You need the id to identify the student in an unique way
- Return a boolean value to indicate that the operation was successful

```
public bool UpdateStudent(int id, string newFirstName, string newLastName)
{
    Student student = GetStudentById(id);
    if (student != null)
    {
        student.FirstName = newFirstName;
        student.LastName = newLastName;
        return true;
    }
    return false;
}
```

ConsoleApp

```
static void Main(string[] args)
{
    university = new University();
    while (true)
    {
        DisplayMenu();

        Console.Write("Your option is: ");
        int option = 0;
        int.TryParse(Console.ReadLine(), out option);

        Console.WriteLine();

        switch (option)
        {
            case 1:
                DisplayAllStudents();
                break;
            case 2:
                ReadStudent();
                break;
            case 3:
                SearchStudentById();
                break;
            case 4:
                DeleteStudentById();
                break;
            case 5:
                UpdateStudentById();
                break;
            case 6:
                return;
            default:
                Console.WriteLine("Invalid option. Try again!");
                break;
        }

        Console.WriteLine();
    }
}
```

Console App (2)

```
class Program
{
    private static University university;

    private static void DisplayMenu()
    {
        Console.WriteLine("What do you want to do?");
        Console.WriteLine("1 - Display all students");
        Console.WriteLine("2 - Add a new student");
        Console.WriteLine("3 - Get a student by id");
        Console.WriteLine("4 - Delete a student by id");
        Console.WriteLine("5 - Update a student");
        Console.WriteLine("6 - Exit");
    }

    private static void DisplayStudent(Student student)
    {
        Console.WriteLine("{0} {1} (id={2}) - {3} - {4}",
            student.FirstName, student.LastName, student.Id, student.Gender, student.BirthDate);
    }

    private static void DisplayAllStudents()
    {
        Console.WriteLine("All students:");
        foreach (Student student in university.GetStudents())
        {
            DisplayStudent(student);
        }
    }
}
```

ConsoleApp (3)

```
private static Student ReadStudent()
{
    Console.Write("Enter first name: ");
    string firstName = Console.ReadLine();

    Console.Write("Enter last name: ");
    string lastName = Console.ReadLine();

    Console.Write("Enter gender (m/f): ");
    string genderLetter = Console.ReadLine().ToLower().Trim();
    Gender gender = Gender.Male;
    switch (genderLetter)
    {
        case "m":
            gender = Gender.Male;
            break;
        case "f":
            gender = Gender.Female;
            break;
        default:
            throw new ArgumentException("Invalid value for gender");
    }

    Console.Write("Enter year of birth (yyyy): ");
    int year = int.Parse(Console.ReadLine());

    Console.Write("Enter month of birth (1-12): ");
    int month = int.Parse(Console.ReadLine());
    Console.Write("Enter day of birth (1-31): ");
    int day = int.Parse(Console.ReadLine());

    return university.AddStudent(firstName, lastName, new DateTime(year, month, day), gender);
}
```

ConsoleApp (4)

```
private static void DeleteStudentById()
    {
        Console.Write("Delete student by id: ");
        int id = int.Parse(Console.ReadLine());
        bool wasStudentDeleted = university.DeleteStudentById(id);
        if (wasStudentDeleted)
        {
            Console.WriteLine("Student was deleted");
        }
        else
        {
            Console.WriteLine("No student with that id");
        }
    }

    private static void SearchStudentById()
    {
        Console.Write("Search student by id: ");
        int id = int.Parse(Console.ReadLine());
        Student foundStudent = university.GetStudentById(id);
        if (foundStudent != null)
        {
            DisplayStudent(foundStudent);
        }
        else
        {
            Console.WriteLine("No student with that id");
        }
    }
```

Interfaces

Interfaces

- An interface represents a contract – a collection of properties, methods, events or indexers.
- In the definition inside the interface, they don’t have access modifiers
- Any class that implements that interface is forced to offer implementations for all the members in that interface
- Since the interface is just a list of definitions, you cannot create an instance of an interface
- An interface can inherit multiple interfaces

```
// Interface from .NET
public interface IComparable
{
    int CompareTo(object obj);
}

// Interface from .NET
public interface IDisposable
{
    void Dispose();
}
```

Interfaces vs Base Classes

- A class can derive from a single class
  - C# doesn’t support multiple inheritance from multiple classes
  - When you derive a class from a base class, the access modifiers for the overridden methods can differ
- A class can implement multiple interfaces
  - When implementing an interface, all methods from that interface have to be implemented in the class and be public

Interfaces – Naming Conventions

- Interface names should be singular nouns (UpperCamelCase)
- Interface names should be by convention prefixed with “I”

Good

```
public interface IDisposable
public interface INotifyPropertyChanged
public interface IMultiValueConverter
```

Evil

```
public interface Disposable
public interface notifyPropertyChanged
public interface multivalueconverter
```

Extracting an interface for StudentsService

```
public interface IStudentsService
{
    Student AddStudent(string firstName, string lastName, DateTime birthDate, Gender gender);
    List<Student> GetStudents();
    Student GetStudentById(int id);
    bool DeleteStudentById(int id);
    bool UpdateStudent(int id, string newFirstName, string newLastName);
}
```

```
public class StudentsService : IStudentsService
{
    // code as in previous slides
}
```

Decoupling the University

- The University class should not be responsible for creating the students service, but it should just assume it will receive and use a service to work with students
- This design pattern is called IoC (Inversion of Control) and sending the service as a parameter in the constructor si called Dependency Injection

```
public class University
{
    private IStudentsService studentsService;

    public University(IStudentsService studentsService)
    {
        this.studentsService = studentsService;
    }

   // code as in previous slides
}
```

```
class Program
{
    static void Main(string[] args)
    {
        university = new University(new StudentsService());
        // code as in previous slides
    }
}
```

Advantages of interfaces and decoupling

- If we want to change the service with another one
  - For example a service that saves and reads students from disk – StudentsFileService
  - All we have to do is to create a class (StudentsFileService) that implements the IStudentsService
  - And replace the creation of the service with the creation of the new class (StudentsFileService)

```
public class University
{
    private IStudentsService studentsService;

    public University(IStudentsService studentsService)
    {
        this.studentsService = studentsService;
    }

   // code as in previous slides
}
```

```
public class StudentsFileService : IStudentsService
{
    // TODO: implement all the methods
}
```

```
class Program
{
    static void Main(string[] args)
    {
        //university = new University(new StudentsService());
        university = new University(new StudentsFileService());
        // code as in previous slides
    }
}
```

What’s next?

- We finished module 01
- Back for module 02 on 09 January

Homework

- Create an console app 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:
  - Implement the entire functionality for this application, similar to the course app
  - Create 2 projects in the same solution: a class library and a console app
  - Sort – try Icomparable
  - You can continue with this homework on the same project as homework from week 04