C# .NET

Laborator 6

Suprascrierea

Overriding

Overriding – suprascrierea

- Este procedeul prin care implementarea unei metode poate fi inlocuita cu o alta implementare intr-o clasa derivata
- Metodele
  - *sealed*
    - keyword, metodele sealed nu mai pot fi suprascrise
  - Virtual
    - Keyword, metodele marcate ca *virtual* pot fi suprascrise in clasaele “copii”
- Metoda “*Tipareste*” din clasa parinte inca exista, ea este doar “ascunsa” de implementarea din clasa derivata
  - *base.method_name()* – va face referire la metoda din clasa *parinte*

```
class Parinte
{
    2 references
    public virtual void Tipareste()
    {
        Console.WriteLine("buna, sunt un parinte");
    }
}
2 references
class Copil : Parinte
{
    2 references
    public override void Tipareste()
    {
        Console.WriteLine("buna, sunt un copil");
    }
}
```

```
Copil copil = new Copil();
copil.Tipareste();
```

Va tipari *“buna, sunt un copil”*

Overriding – suprascrierea

Va tipari

“buna, sunt un parinte”

“buna, sunt un copil”

```
class Parinte
{
    3 references
    public virtual void Tipareste()
    {
        Console.WriteLine("buna, sunt un parinte");
    }
}
2 references
class Copil : Parinte
{
    3 references
    public override void Tipareste()
    {
        base.Tipareste();
        Console.WriteLine("buna, sunt un copil");
    }
}

Copil copil = new Copil();
copil.Tipareste();
```

Overriding – suprascrierea

- O metoda virtuala poate fi suprascrisa oridecateori este nevoie
- Apelul inspre metodele de baza la inceputul unei metode care suprascrie – good practice.
- Liskov Substitution Principle (LSP)
  - objects of a superclass should be replaceable with objects of its subclasses without breaking the application.

```
class Parinte
{
    5 references
    public virtual void Tipareste()
    {
        Console.WriteLine("buna, sunt un parinte");
    }
}
3 references
class Copil : Parinte
{
    5 references
    public override void Tipareste()
    {
        base.Tipareste();
        Console.WriteLine("buna, sunt un copil");
    }
}
0 references
class Nepot : Copil
{
    5 references
    public override void Tipareste()
    {
        base.Tipareste();
        Console.WriteLine("buna, sunt un nepot");
    }
}
```

- Un obiect din subclasa poate fi persistat intr-oo referinta de tipul superclasei
- Apelul unei metode suprascrise
  - Intotdeauna din subclasa care suprascrie
- Ce va tipari?

```
static void Main(string[] args)
{

    Parinte parinte = new Parinte();
    Parinte copil = new Copil();
    Parinte nepot = new Nepot();

    parinte.Tipareste();
    copil.Tipareste();
    nepot.Tipareste();
}
```

```
class Parinte
{
    5 references
    public virtual void Tipareste()
    {
        Console.WriteLine("buna, sunt un parinte");
    }
}
3 references
class Copil : Parinte
{
    5 references
    public override void Tipareste()
    {
        base.Tipareste();
        Console.WriteLine("buna, sunt un copil");
    }
}
0 references
class Nepot : Copil
{
    5 references
    public override void Tipareste()
    {
        base.Tipareste();
        Console.WriteLine("buna, sunt un nepot");
    }
}
```

```
static void Main(string[] args)
{

    Parinte parinte = new Parinte();
    Parinte copil = new Copil();
    Parinte nepot = new Nepot();

    parinte.Tipareste();
    copil.Tipareste();
    nepot.Tipareste();
}
```

- Parinte
  - “buna, sunt un parinte”
- Copil
  - “buna, sunt un parinte”
  - “buna, sunt un copil”
- Nepot
  - “buna, sunt un parinte”
  - “buna, sunt un copil”
  - “buna, sunt un nepot”

```
class Parinte
{
    5 references
    public virtual void Tipareste()
    {
        Console.WriteLine("buna, sunt un parinte");
    }
}
3 references
class Copil : Parinte
{
    5 references
    public override void Tipareste()
    {
        base.Tipareste();
        Console.WriteLine("buna, sunt un copil");
    }
}
0 references
class Nepot : Copil
{
    5 references
    public override void Tipareste()
    {
        base.Tipareste();
        Console.WriteLine("buna, sunt un nepot");
    }
}
```

Method hiding

```
class Parinte
{
    1 reference
    public  void Tipareste()
    {
        Console.WriteLine("buna, sunt parinte");
    }
}
3 references
class Copil : Parinte
{
    1 reference
    public void Tipareste()
    {
        Console.WriteLine("buna, sunt copil");
    }
}
```

void Copil.Tipareste()

CS0108: 'Copil.Tipareste()' hides inherited member 'Parinte.Tipareste()'. Use the new keyword if hiding was intended.

Show potential fixes (Ctrl+.)

- Lipsa tandemului *virtual* / *override*
  - Ascunderea metodei

```
Copil copil1 = new Copil();
Parinte copil2 = new Copil();

copil1.Tipareste(); // buna sunt un copil
copil2.Tipareste(); // buna sunt un parinte
```

Method hiding

```
class Parinte
{
    1 reference
    public  void Tipareste()
    {

    }
}
3 references
class Co
{
    1 refere
    publi
    {


        Con
    }
}
```

- Lips *at / override*
  - As etodei

```
Copil copil1 = new Copil();
Parinte copil2 = new Copil();

copil1.Tipareste(); // buna sunt un copil
copil2.Tipareste(); // buna sunt un parinte
```

sealed

- La suprascriere putem marca “*sealed*” metoda
  - In clasele care mostenesc “Copil” metoda *sealed* nu va mai putea fi suprascirsa
- O clasa marcata *sealed*
  - Nu mai poate fi mostenita de o alta clasa

```
class Parinte
{
    5 references
    public virtual void Tipareste()
    {
        Console.WriteLine("buna, sunt un parinte");
    }
}
3 references
class Copil : Parinte
{
    4 references
    public sealed override void Tipareste()
    {
        base.Tipareste();
        Console.WriteLine("buna, sunt un copil");
    }
}
0 references
class Nepot : Copil
{
    5 references
    public override void Tipareste()
    {
        base.Tipareste();
        Console.WriteLine("buna,
    }
}
```

void Nepot.Tipareste()

CS0239: 'Nepot.Tipareste()': cannot override inherited member 'Copil.Tipareste()' because it is sealed

Overriding, overloading

- Overriding
  - supraSCRIEREA
  - implementarea unei metode poate fi inlocuita cu o alta implementare intr-o clasa derivata
- overloading
  - supraINCARCAREA
  - Mai multe metode cu acelasi nume dar cu *semnatura* diferita
    - Semnatura – numar parametrii, tip pararmetrii
- Metodele virtuale din *parinte* pot fi supraincarcate in *subclase*

```
class Parinte
{
    3 references
    public virtual void Tipareste()
    {
        Console.WriteLine("buna, sunt un parinte");
    }
}
2 references
class Copil : Parinte
{
    3 references
    public override void Tipareste()
    {
        base.Tipareste();
        Console.WriteLine("buna, sunt un copil");
    }
    0 references
    public void Tipareste(string interlocutor)
    {
        Console.WriteLine($"buna {interlocutor}, sa stii ca sunt un copil");
    }
}
```

Object class

Object

- Clasa Object
  - Parintele tuturor claselor C#
    - Constructor no-arg

```
Object o = new Object();
```

- Implementeaza o serie de functii de baza care pot fi suprascrise
  - ToString( )
    - transforma un obiect in string!
    - Va fi apelata oridecateori obiectul trebuie tratat ca stirng
      - Concatenari, interpolari, string-uri verbatim
      - Console.writeline
      - etc

Properties override

Properties override

- Proprietatile
  - Sunt mostenite cu aceleasi reguli ca si oricare alti membrii
  - Virtual/override
    - la nivel de proprietate nu de accessor
    - Nu putem marca “virtual” sau “override” doar setter-ul sau getter-ul

Constructors

inheritance

Inheritance – constructors

- Constructori
  - Daca in clasa parinte avem definiti constructori iar constructorul *noarg* nu este definit, atunci toti constructorii claselor *derivate*
  - Apelul ctor-ului parinte :
    - cuvantul cheie *base*
    - Base – ctor-ului parintelui din nivelul imediat superior (parinte nu bunic)
  - Pot fi supraincarcati fie in parinte fie in unul din fii
  - Parametrii constructorului *base* pot fi *hardcode-ati*

```
class Vagon
{
    private string serie="";
    private int masa=2400;
    2 references
    public Vagon(string serie, int masa)
    {
        this.serie = serie;
        this.masa = masa;
    }
}
6 references
class VagonDeMarfa : Vagon
{
    1 reference
    public VagonDeMarfa(string seria):base(seria,2500)
    {
    }
    1 reference
    public VagonDeMarfa(string seria, int masa):base(seria,masa)
    {
    }
}
1 reference
class VagonDeSfecla : VagonDeMarfa
{
    0 references
    public VagonDeSfecla(string seria):base(seria)
    {
    }
}
3 references
class VagonDePietris : VagonDeMarfa
{
    1 reference
    public VagonDePietris(string seria) : base(seria,1000)
    {
    }
}
```

Inheritance – constructors

- Constructorul *no-arg*
- Daca a fost definit in clasa parinte
  - va fi apelat in mod implicit
  - Nu este nevoie de apelul *base()*
- Daca nu a fost definit in clasa parinte
  1. Clasa parinte nu are definit niciun alt constructor – va fi apelat **implicit**
  2. Clasa parinte are definiti alti construci – constructorul no-arg nu mai poate fi apelat

```
class Vagon
{
    private string serie="";
    private int masa=2400;
    2 references
    public Vagon(string serie, int masa)
    {
        this.serie = serie;
        this.masa = masa;
    }
}
6 references
class VagonDeMarfa : Vagon
{
    1 reference
    public VagonDeMarfa(string seria):base(seria,2500)
    {
    }
    1 reference
    public VagonDeMarfa(string seria, int masa):base(seria,masa)
    {
    }
}
1 reference
class VagonDeSfecla : VagonDeMarfa
{
    0 references
    public VagonDeSfecla(string seria):base(seria)
    {
    }
}
3 references
class VagonDePietris : VagonDeMarfa
{
    1 reference
    public VagonDePietris(string seria) : base(seria,1000)
    {
    }
}
```

Inheritance – initalization order

- Clasa "*derivata*"
  - ***Este*** de tipul clasa parinte
- Clasa "parinte" va fi initialiata prima
  - Regulile clasice de initializare a claselor
    1. Campuri
    2. Corpul constructorilor

```
class Vagon
{
    private string serie="";
    private int masa=2400;
    1 reference
    public Vagon(string serie, int masa)
    {
        this.serie = serie;
        this.masa = masa;
    }
}
3 references
class VagonDeMarfa : Vagon
{
    int capacitate = 0;

    1 reference
    public VagonDeMarfa(string seria, int masa):base(seria,masa)
    {
        this.capacitate = masa * 12;
    }
}

3 references
class VagonDePietris : VagonDeMarfa
{
    private string culoare;
    1 reference
    public VagonDePietris(string seria, string culoare) : base(seria,1000)
    {
        this.culoare = culoare;
    }
}

VagonDePietris vpe = new ctori.VagonDePietris("188773hh4hi9","alb");
```

Inheritance – initalization order

- Clasa "*derivata*"
  - ***Este*** de tipul clasa parinte
- Clasa "parinte" va fi initialiata prima
  - Regulile clasice de initializare a claselor
    1. Campuri
    2. Corpul constructorilor

```
class Vagon
{
    private string serie="";
    private int masa=2400;
    1 reference
    public Vagon(string serie, int masa)
    {
        this.serie = serie;
        this.masa = masa;
    }
}
3 references
class VagonDeMarfa : Vagon
{
    int capacitate = 0;

    1 reference
    public VagonDeMarfa(string seria, int masa):base(seria,masa)
    {
        this.capacitate = masa * 12;
    }
}

3 references
class VagonDePietris : VagonDeMarfa
{
    private string culoare;
    1 reference
    public VagonDePietris(string seria, string culoare) : base(seria,1000)
    {
        this.culoare = culoare;
    }
}
VagonDePietris vpe = new ctori.VagonDePietris("188773hh4hi9","alb");
```

1 2 3 4 5 6 7 0

Static

modificator

static

- Exercitiu
- Un student are nume, prenume si un cnp. Scrieti un program care va Numara cati student au fost instantiati

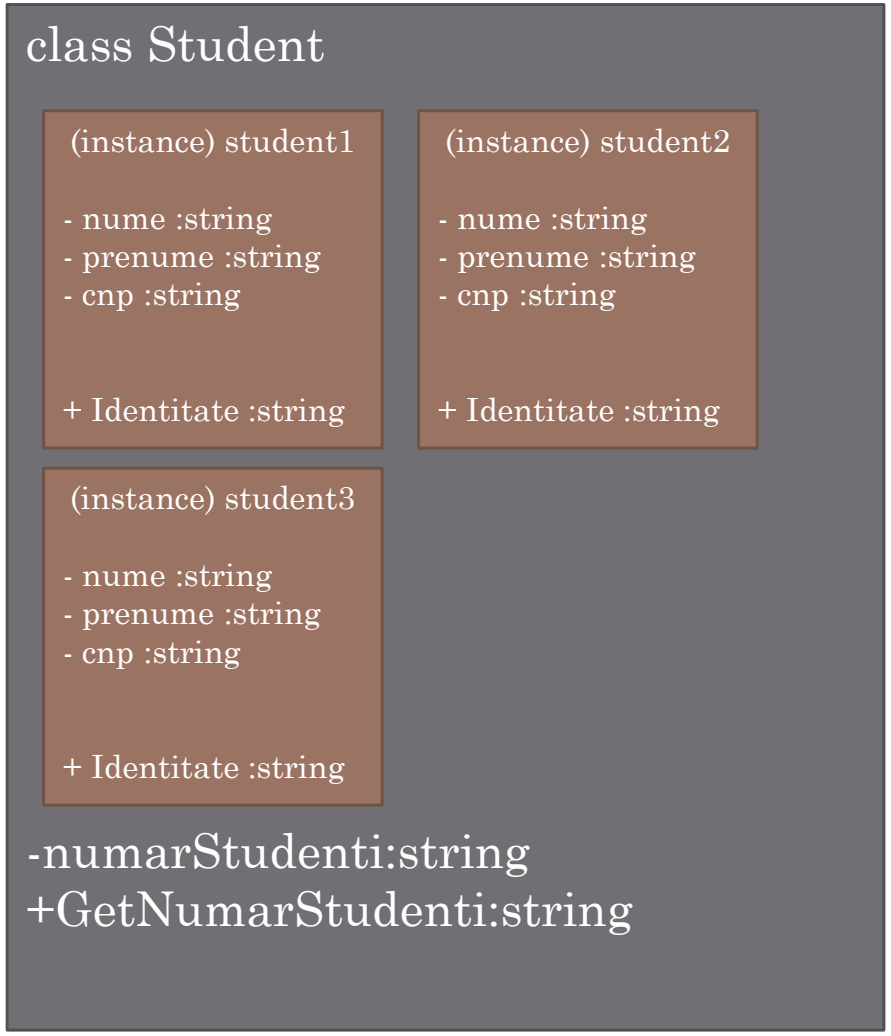
static

- static – modificator ce marchează un membru al **CLASEI**
- Nume,prenume,cnp – membri ai **instantei**, ai **obiectului.**
- Membri statici
  - Sunt specifici clasei
  - Sunt independenți fata de obiecte
- Clasa – unica la nivelul intregii aplicatii
  - Membrii statici – unici la nivelul intregii aplicatii

```
class Student
{
    private string cnp;
    private string nume;
    private string prenume;

    private static int numarStudenti = 0;
    3 references
    public Student(string nume, string prenume, string cnp)
    {
        this.cnp = cnp;
        this.prenume = prenume;
        this.nume = nume;
        Student.numarStudenti++;
    }
    0 references
    public string Identitate
    {
        get
        {
            return $"Nume: {nume}, Prenume:{prenume}, DateBuletin:{ cnp}";
        }
    }
    1 reference
    public static int GetNumarStudenti()
    {
        return Student.numarStudenti;
    }
}
```

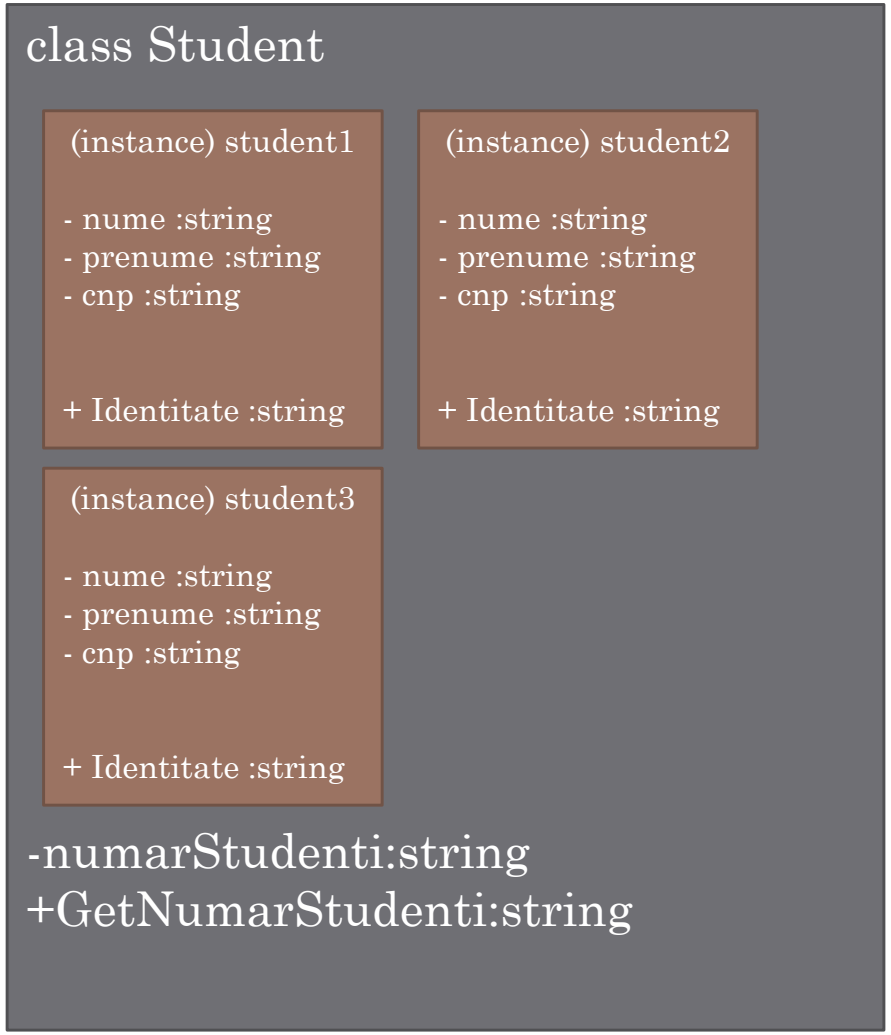
static



- Membrii statici (numarStudenti) sunt accesibili din interiorul metodelor instantelor?

- Membrii instantelor (nume, Identitate) sunt accesibili din interiorul unei metode statice?

static



- Membrii static (numarStudenti) sunt accesibili din interiorul metodelor instantelor?
- *DA, chiar si cei private!*

- Membrii instantelor (nume, Identitate) sunt accesibili din interiorul unei metode statice?
- *NU. Clasa nu poate determina carui obiect/instanta ii apartine membrul apelat*

static – accesarea membrilor

- Din exteriorul clasei:
  - NumeleClasei.numeleMembruluiPublicAlClasei

```
static void Main(string[] args)
{

    Student student1 = new Student("popescu", "george", "12333344");
    Student student2 = new Student("popescu", "george", "12333344");
    Student student3 = new Student("popescu", "george", "12333344");

    // va afisa 3
    Console.WriteLine(Student.GetNumarStudenti());
```

- Din interiorul clasei
  - Prefixul *NumeleClasei* este optional

```
public static int GetNumarStudenti()
{
    return Student.numarStudenti;
}
```

```
public static int GetNumarStudenti()
{
    return numarStudenti;
}
```

- Campuri statice
- Metode statice
- Proprietati statice

```
private static int numarStudenti = 0;

/// <summary>
/// Numarul total de studenti
/// </summary>
/// <returns></returns>
1 reference
public static int GetNumarStudenti()
{
    return numarStudenti;
}
/// <summary>
/// Numarul total al baietilor
/// </summary>
1 reference
public static int NumarBaieti { private set; get; }

3 references
public Student(string nume, string prenume, string cnp)
{
    this.cnp = cnp;
    this.prenume = prenume;
    this.nume = nume;
    numarStudenti++;
    if (cnp.StartsWith('1') || cnp.StartsWith('5'))
    {
        NumarBaieti++;
    }
}
```

static

Membrii obiectului (instanţei)

- Campurile obiectului
  - Aparţin obiectului (instanţei)
  - Au valori diferite de la un obiect la altul
- Metodele obiectului
  - Aparţin obiectului
  - Pot accesa membrii (campuri, metode, proprietăţi) ale obiectului
  - Pot accesa campuri statice
- Proprietăţi ale instanţei
  - Aparţin obiectului
  - Au valori diferite de la un obiect la altul
  - Pot accesa membrii (campuri, metode, proprietăţi) ale obiectului
  - Pot accesa campuri statice

Membrii statici

- Campuri statice
  - Aparţin clasei
  - Au aceeaşi valoare pentru toate obiectele
- Metodele clasei
  - Aparţin clasei
  - **NU** pot accesa membrii (campuri, metode, proprietăţi) ale obiectului
  - Pot accesa campuri statice
- Proprietăţi statice
  - Aparţin clasei
  - Au aceeaşi valoare pentru toate obiectele
  - **NU** pot accesa membrii (campuri, metode, proprietăţi) ale obiectului
  - Pot accesa campuri statice

static

- Cand folosim membri statici
  - Când lucram cu date care nu sunt specifice unei instanţe
  - Metode – când metoda nu are nevoie de niciun membru al obiectului.

Clasa statica

- Poate conţine doar membri statici

- Utilizari
  - Colecţii de funcţii de sine stătătoare
    - ex clasa Math, Array
    - Clase *util* care operează asupra unor obiecte definite de noi
  - Statistici la nivel de aplicaţie
    - ex clasa ThreadPool
  - Colecţii de valori constante

```
/// <summary>
/// Registrul interplanetar al studentilor
/// </summary>
0 references
static class RegistruStudenti
{
    private static int numarStudenti;
    0 references
    public static int GetNumarStudenti()
    {
        return numarStudenti;
    }
    0 references
    public static void IncrementeazaNumarulStudentilor()
    {
        numarStudenti++;
    }

    0 references
    public void Tipareste() {

        /// eroare - clasa statica nu poate contine membri de instanta, ne-statici
        Console.WriteLine(numarStudenti);
    }
}
```

const

const

- Keyword care defineste o constanta
- PascalCase
- Valorile sunt stabilite la **compilare**
- Performant
- Functioneaza corect doar cu value types
  - Cu ref types, valoarea trebuie sa fie *null*
- Se comporta ca si *static readonly*

```
class Mig21LanceR
{
    public const int Echipaj = 1;
    private const string Vechime = "antic";
    protected const double VitezaMaxima = 2876.4;
    private const int[] x = null;
}
```

const

const

- Valorile nu pot fi modificate
- Valorile sunt stabilite la **compilare**
- Performant
- Functioneaza doar cu value types si string-uri

static readonly

- Valorile nu pot fi modificate
- Valorile sunt stabilite la **run-time**
- Mai putin performant
- Functioneaza atat cu value types cat si cu reference types

Clase abstracte

Abstract classes – clase abstracte

- Clase care nu pot fi instantiate
  - Constructorii nu pot fi apelati
  - Nu pot fi creeate obiecte- instanta ale claselor abstracte
- Pot contine
  - Campuri
  - Constructori
  - Proprietati
  - Metode
  - **Metode abstracte**
    - Metode care nu au o implementare
    - Trebuie implementate in mod obligatoriu in orice subclasa **non-abstarcta**

modificatorAcces ***abstract*** *returnType* numeFunctie(params…);

```
abstract class ProdusElectronic
{
    1 reference
    public ProdusElectronic(int pret)
    {
        Pret = pret;
    }
    1 reference
    public int Pret { get; private set; }

    // aceasta este o metoda abstracta
    1 reference
    public abstract void Porneste();
}

1 reference
class TV : ProdusElectronic
{
    0 references
    public TV() : base(1500)
    {
    }
    1 reference
    public override void Porneste()
    {
        Console.WriteLine("Am pornit televizorul");
    }
}
```

Abstract classes – clase abstracte

- Metodele abstracte
  - **Obliga** subclasele non-abstracte sa ofere o implementare
  - **Override**
    - de fapt este vorba de o suprascriere
    - Metodele *abstract* sunt by default *virtual*
  - Nu pot fi private - nu ar fi vizibile in subclasa
    - Doar *public, protected*
- Clasa abstracta
  - Nu poate fi sealed (trebuie extinsa pentru a putea fi instantiata)
  - UML
    - Numele clasei abstracte – reprezentata cu *Italic*
    - Metodele abstracte – reprezentate cu *Italic*

```
abstract class ProdusElectronic
{
    1 reference
    public ProdusElectronic(int pret)
    {
        Pret = pret;
    }
    1 reference
    public int Pret { get; private set; }

    // aceasta este o metoda abstracta
    1 reference
    public abstract void Porneste();
}

1 reference
class TV : ProdusElectronic
{
    0 references
    public TV() : base(1500)
    {
    }
    1 reference
    public override void Porneste()
    {
        Console.WriteLine("Am pornit televizorul");
    }
}
```

Namespaces

Namespace

- Un container prin care organizam *entitatile*
  - *clase*
  - *enum*
  - *interfete*
  - *alte namespace-uri*
  - *etc*
- Definesc un *scope* in care entitatile au sens
- **PascalCase**

```
namespace Laborator9
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {

        }
    }
}
```

Namespace

- Face parte din numele complet ale entitatilor
  - Program
    - Laborator9.Program
- Namespace-ul poate imbrica alte namespace-uri
  - Laborator9.Students
  - Numele clasei : Laborator9.Students.Student

```
namespace Laborator9
{
    1 reference
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Laborator9.Program program;
            Laborator9.Students.Student student = new Students.Student();
        }
    }
    namespace Students
    {
        2 references
        class Student {
        }
    }
}
```

Namespace

- Permite doua entitati cu acelasi nume in acelasi assembly
  - In namespace-uri diferite
  - Vor avea nume identice dar nume complete diferite

```
namespace Laborator9
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Laborator9.Highschool.Student highschoolStudent =
                new Laborator9.Highschool.Student(11);
            Laborator9.University.Student univeristyStudent =
                new Laborator9.University.Student(2,"Informatica");
        }
    }
    namespace Highschool
    {
        3 references
        class Student {
            1 reference
            public Student(int clasa)
            {
            }
        }
    }
    namespace University
    {
        3 references
        class Student
        {
            1 reference
            public Student(int anul, string specializarea)
            {
            }
        }
    }
}
```

Namespace

- Numele complete – adresate relative
- *Highschool* si *University* sunt namespace-uri imbricate in namespace-ul *Laborator9*
  - Prefixul *Laborator9* este optional in contextul root-ului namespace-ului

```
namespace Laborator9
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Laborator9.Highschool.Student highschoolStudent =
                new Laborator9.Highschool.Student(11);
            Laborator9.University.Student univeristyStudent =
                new Laborator9.University.Student(2,"Informatica");
        }
    }
    namespace Highschool
    {
        3 references
        class Student {
            1 reference
            public Student(int clasa)
            {
            }
        }
    }
    namespace University
    {
        3 references
        class Student
        {
            1 reference
            public Student(int anul, string specializarea)
            {
            }
        }
    }
}
```

Namespace

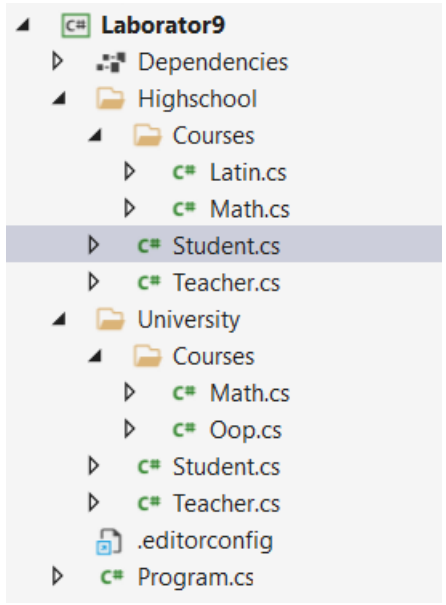
- clauza “using”
  - Aduce contextul namespace-ului in contextul current
  - Entitatile din namespace-ul importat precum si din namespace-urile imbricate nu mai trebuie prefixate cu namespace-ul importat prin “*using*”

```
using System;
using Highschool;

namespace Laborator9
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Student highschoolStudent = new Student(11);
        }
    }
}
```

Nume complet:
Laborator9.Highschool.Student

Namespace-uri



- Simplifica organizarea codului
- Namespaces
  - Reflecta folder structure
- R. click -> New folder
  - Clase adaugate – namespace automat

```
namespace Laborator9.Highschool.Courses
{
    0 references
    class Latin
    {
    }
}
```

```
namespace Laborator9.Highschool
{
    3 references
    class Student
    {
        1 reference
        public Student(int clasa)
        {
        }
    }
```

```
namespace Laborator9
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Highschool.Student student = new Highschool.Student(11);
            Highschool.Courses.Latin latinCourse = new Highschool.Courses.Latin();
        }
    }
}
```

Tools

- Diagrame UML – click AICI

Va multumesc!