C# .NET

Laborator 5

Supraincarcarea constructorului

- Constructorii pot fi supraincarcati
- Compilatorul va alege constructorul corespunzator la fel ca si in cazul metodelor – **cum**?

```
class Student
{
    public string nume;
    public string prenume;
    public int varsta;
    public int[] note;

    0 references
    public Student()
    {
        this.note = new int[10];
    }
    0 references
    public Student(string nume)
    {
        this.note = new int[10];
        this.nume = nume;
    }
    0 references
    public Student(string nume, string prenume)
    {
        this.note = new int[10];
        this.nume = nume;
        this.prenume = prenume;
    }
    1 reference
    public Student(string nume, string prenume, int varsta)
    {
        this.note = new int[10];
        this.nume = nume;
        this.prenume = prenume;
        this.varsta = varsta;
    }
}
```

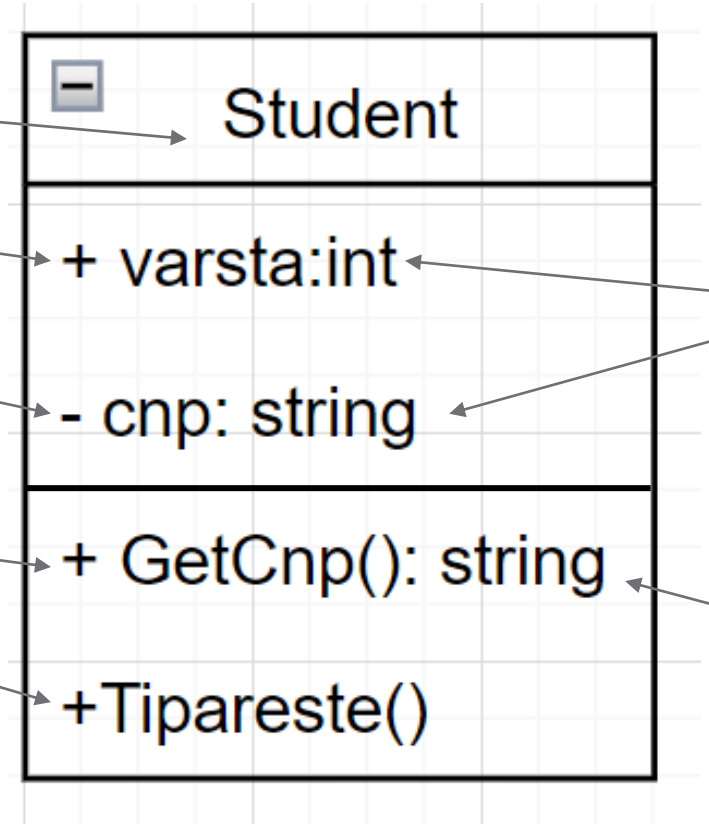
- Constructorii se pot chema intre ei
  - :this(/*lista parametri*/)

```
class Student
{
    public string nume;
    public string prenume;
    public int varsta;
    public int[] note;

    1 reference
    public Student()
    {
        this.note = new int[10];
    }
    1 reference
    public Student(string nume):this()
    {
        this.nume = nume;
    }
    1 reference
    public Student(string nume, string prenume) : this(nume)
    {
        this.prenume = prenume;
    }
    1 reference
    public Student(string nume, string prenume, int varsta) : this(nume, prenume)
    {
        this.varsta = varsta;
    }
}
```

Relatii intre obiecte

UML – campuri si metode

- Nume clasa
- Camp public
- Camp privat
- metode

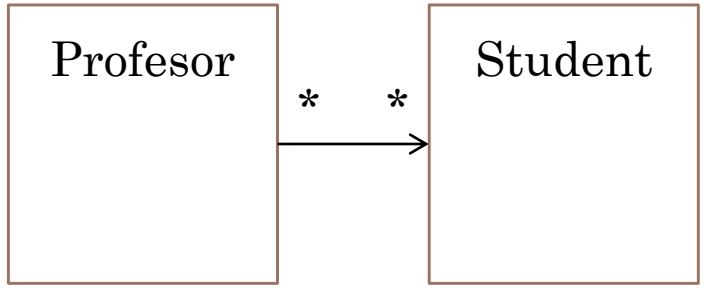


Tools

- Diagrame UML – click AICI

Asociere

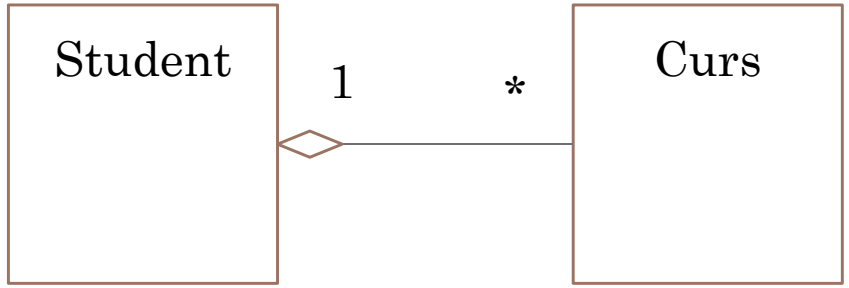
- UML – universal modeling language
- Un obiect poate apela un alt obiect
  - O relatie de tipul "*has a*"
  - Poate accesa campuri
  - Poate invoca metode ale celuilalt obiect
  - Relatia reprezentata de o sageata
  - Cele doua obiecte pot exista independent
- Ex, Profesorul ii acorda o nota Studentului
  - Exercitiu : implementati



Agregare

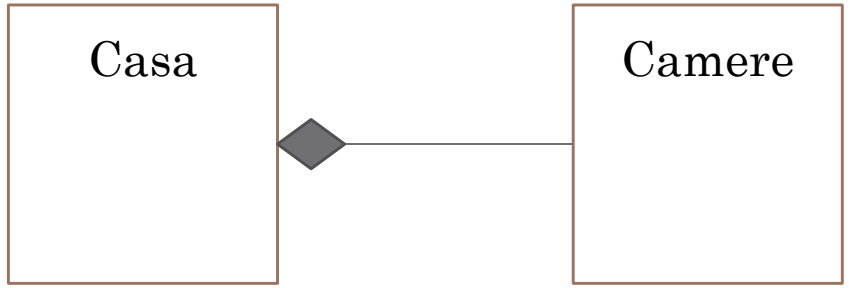
- Un tip de asociere
  - O relatie de tipul "*has a*"
  - Cele doua obiecte pot exista independent
  - Romb gol
  - Poate accesa campuri
  - Poate invoca metode ale celuilalt obiect

- Ex, Studentul participa la mai multe Cursuri
  - Exercitiu :
    - Afisare student
    - Afisare curs
    - Afisati cursurile unui student



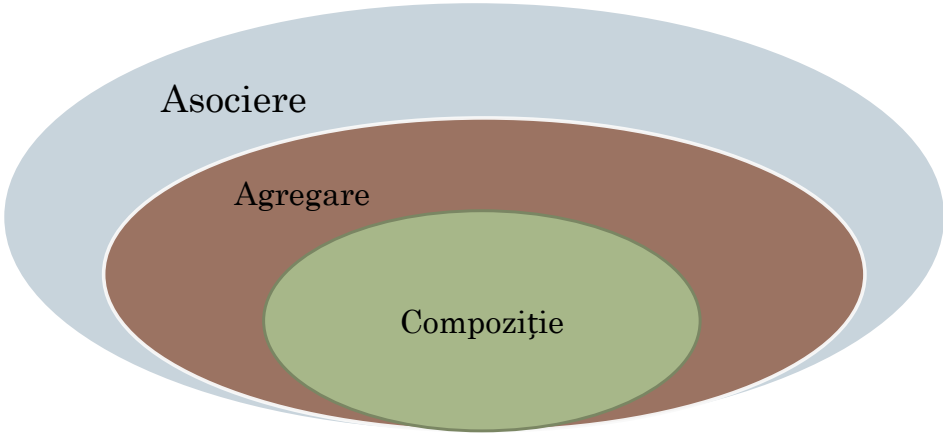
Compozitie

- Cea mai puternica relatie de asociere
- Un obiect detine ca si camp, o referinta spre un alt obiect si nu poate functiona fara acel obiect, existenta acestuia neavand sens fara obiectele din care este compus
- Exemple:
  - Casa-camere
  - Motor – pistoane
  - Om-Creier



Relații între obiecte

- Asociere
  - Relația reprezentata de o săgeata
- Agregare
  - Un tip de asociere
    - Cele doua obiecte pot exista independent
    - Romb gol 
  - Un obiect poate apela un alt obiect
    - Poate accesa câmpuri
    - Poate invoca metode ale celuilalt obiect



  - Ex, Studentul participa la mai multe Cursuri si **persista** cate o referința a fiecărui curs la care participa
- Compoziție 
  - Un tip de agregare
  - Cea mai puternica relație de asociere
  - Un obiect deține ca si câmp, o referința spre un alt obiect si nu poate funcționa fără acel obiect
  - Romb plin
  - Exemple:
    - Casa-camere
    - Motor – pistoane
  - Motorul este COMPUS din pistoane si NU POATE FUNCTIONA fără ele

Exercitii

- Scrieti un program care va modela un apartament.
- Un **apartament** este caracterizat de o **Adresa** si un numar de **camere**.
  - Fiecare camera este caracterizata de o lungime, latime si intaltime precum si de un nume.
    - Camera va contine o metoda “GetDescription” care va returna numele camerei.
  - Adresa contine strada, numarul, scara si etajul, numarApartament.
    - Clasa adresa va contine o metoda “GetDescription” care va returna adresa sub forma “strada, numarul, scara, etajul, numarApartament”.
  - Apartamentul va contine urmatoarele metode
    - Tiparire – va afisa descrierea camerelor precum si adresa apartamentului
    - O metoda care va returna suprafata totala a apartamentului.

Rulati programul, efectuati **diagrama UML** a acestuia

Single responsibility principle

Single responsibility principle

- Single responsibility principle

- The single-responsibility principle (SRP) is a computer-programming principle that states that every module, **class** or function in a computer program should have **responsibility over a single part** of that program's functionality, and it should **encapsulate** that part.
  - **responsibility over a single part**
    - o functie trebuie sa faca un singur lucru
    - o clasa trebuie sa modeleze un singur tip de obiecte
      - metodele unei clase trebuie sa opereze asupra propriilor campuri
        - *Inappropriate intimacy*: code smell: a class that has dependencies on implementation details of another class

  - **Encapsulate**
    - Executia unei functii trebuie sa depinda doar de parametrii acesteia si nu de alte date “ocultе”
    - O clasa trebuie sa functioneze de sine statatoare. Cu exceptia elementelor aflate in relatie de compozitie, functionarea corecta a clasei nu trebuie sa depinda de actiuni externe
      - Ex: citirea **prin consola** notelor studentului intr-o metoda a clasei
        - Clasa depinde de existenta consolei si nu poate fi folosita in niciun context in care aceasta nu exista, spre ex: server web, desktop app, etc
    - Sanity checks – verificam daca parametrii sunt “sanatosi la cap”

Clasele - impartirea

- Cum impartim programul in clase?
- OOP – modeleaza cat mai bine lumea exterioara in contextual dat
- Impartim problema in ACTORI. Actori = obiecte, clase
  - Fiecare clasa va modela un singur tip de entitate.
- Atentie la responsabilitati
  - Este responsabilitatea soferului sa porneasca motorul masinii? Soferul va porni doar masina…
  - Este responsabilitatea motorului sa verifice cheia?
- Atentie la complexitate: nu adaugam complexitate in plus in mod inutil
  - Pentru un sofer, masina contine usa, volan, schimbator, pedale
  - Pentru un mecanic reprezinta mult mai multe
  - Pentru un pieton…. claxonul

Enumerații

enum

- Definirea unui set finit si discret de valori
- Cuvant cheie: enum
  - Identificatorul enumeratiei
    - PascalCase
  - Valorile
    - PascalCase
- Value type
- Declarare si initializare:

```
TipAutovehicul tip = TipAutovehicul.Diesel;
```

```
enum ZileleSaptamanii
{
    Luni,
    Marti,
    Miercuri,
    Joi,
    Vineri,
    Sambata
}
0 references
enum TipAutovehicul
{
    Electric,
    Hibrid,
    Diesel,
    Benzina,
    GPL
}
```

enum

- In spatele valorilor se afla cate un intreg
- Valori implicite
  - 0….n
- Valori explicite
  - Nu trebuie sa fie neaparat in ordine
    - De ce ai face asa ceva?!
  - Nu trebuie asignate valori numerice tuturor elementelor enumeratiei
    - Ce valori vor fi atribuite implicit elementelor lipsa?

```
enum ZileleSaptamanii
{
    Luni = 0,
    Marti = 1,
    Miercuri = 2,
    Joi = 3,
    Vineri = 4,
    Sambata = 5
}
```

enum – citire, afisare

```
enum TipAutovehicul
{
    Electric,
    Hibrid,
    Diesel,
    Benzina,
    GPL
}
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {

        TipAutovehicul valoareEnum  = (TipAutovehicul)Enum.Parse(typeof(TipAutovehicul),"Benzina");

        Console.WriteLine(valoareEnum);
        Console.WriteLine((int)valoareEnum);
    }
}
```

Ce va afisa programul de mai sus?

Proprietati

Properties

Accesul la membrii privați - Getter / Setter

- Pot avea orice modificatori de acces

- Public getter, Private setter
  - Permite modificarea valorii doar din interiorul clasei
  - Controlul asupra valorii este strict in interiorul clasei

- Exercițiu:
  - Scrieți o clasa care sa modeleze un cont bancar. Contul va permite depunerea, extragerea numerarului precum si afișarea soldului.

```
class Student {

    private string cnp;
    1 reference
    public string GetCnp()
    {
        return cnp;
    }
    0 references
    private void SetCnp(string newCnp)
    {
        cnp = newCnp;
    }
}
```

Properties

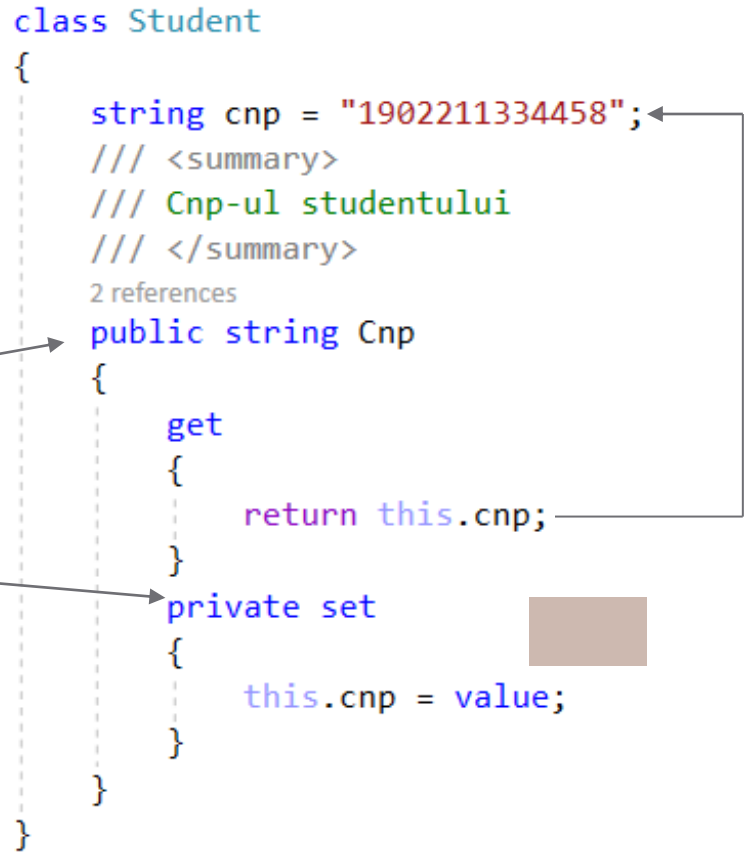
```
class Student {

    private string cnp;
    1 reference
    public string GetCnp()
    {
        return cnp;
    }
    0 references
    private void SetCnp(string newCnp)
    {
        cnp = newCnp;
    }
}
```

```
class Student
{
    string cnp = "1902211334458";
    /// <summary>
    /// Cnp-ul studentului
    /// </summary>
    2 references
    public string Cnp
    {
        get
        {
            return this.cnp;
        }
        private set
        {
            this.cnp = value;
        }
    }
}
```

Properties

<modificatorAcces> <tipul> Numele

- Înlocuiesc conceptul de getter/setter
- Reprezinta una sau mai multe **METODE**
  - Compilatorul genereaza doua metode, *get_Cnp*, *set_Cnp*
- PascalCase
- Modificatori de acces
  - Modificator de acces al proprietății
    - Se aplica in cazul de fata get-ului
  - Modificator mai restrictiv
- Return
  - Valoarea returnata
    - Trebuie sa corespunda tipului
- Value
  - keyword
  - Valoarea – parametrul setter-ului



Properties - usage

```
static void Main(string[] args)
{
    Student student = new Student();

    student.Cnp = "newCnp";

    Console.WriteLine(student.Cnp);
}
```

- Funcţiile getter/setter vor fi tratate precum câmpurile clasice
- Modificatorii de acces funcţionează in continuare

```
class Student
{
    string cnp = "1902211334458";
    /// <summary>
    /// Cnp-ul studentului
    /// </summary>
    2 references
    public string Cnp
    {
        get
        {
            return this.cnp;
        }
        private set
        {
            this.cnp = value;
        }
    }
}
```

Auto-initialized properties

```
class Student
{
    /// <summary>
    /// CNP-ul studentului
    /// </summary>
    2 references
    public string Cnp { get; private set; } = "1321adsdsadasdas";
}
```

- Se comporta ca un câmp de sine stătător
- Getter/setter cu access modifiers diferiți

```
class Student
{
    string cnp = "1902211334458";
    /// <summary>
    /// Cnp-ul studentului
    /// </summary>
    2 references
    public string Cnp
    {
        get
        {
            return this.cnp;
        }
        private set
        {
            this.cnp = value;
        }
    }
}
```

Properties

```
class Student
{
    private int[] note;

    0 references
    public double Media
    {
        get
        {
            if (note == null || note.Length == 0)
            {
                return 0;
            }
                double media = 0;
            foreach(var nota in note)
            {
                media += nota;
            }
            return media / note.Length;
        }
    }
}
```

- getter only

```
class User
{
    private string password;

    0 references
    public string Password
    {
        set
        {
            this.password = value;
        }
    }
}
```

- setter only

Properties

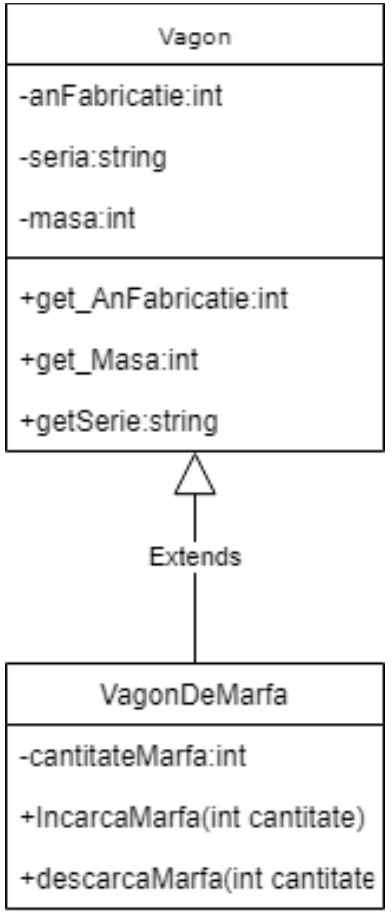
- Avantaje
  - Toate avantajele getter/setter
    - Ascunderea implementării
    - Extendabilitate
    - Mentenabilitate
  - Lizibilitate
    - Mai usor de lucrat decat cu functii get/set
      - Codul este mai „curat”, mai usor de citit
    - Incapsulare – implementata *mai* „corect”
      - Functiile get/set expuneau date/starea sub forma behaviour
- Dezavantaje
  - Nu exista un standard pt reprezentarea UML a proprietatilor
- Cand nu se folosesc
  - Private – campuri clasice

Mostenirea

OOP

Moștenirea - inheritance

- Procedura prin care o clasa moștenește campurile si metodele unei alte clase.
  - Campuri, metode, proprietăţi
- Clasa "*fiu*"/subclasa poate adauga functionalitate clasei parinte/superclasei :
  - **Extinde** (extends)

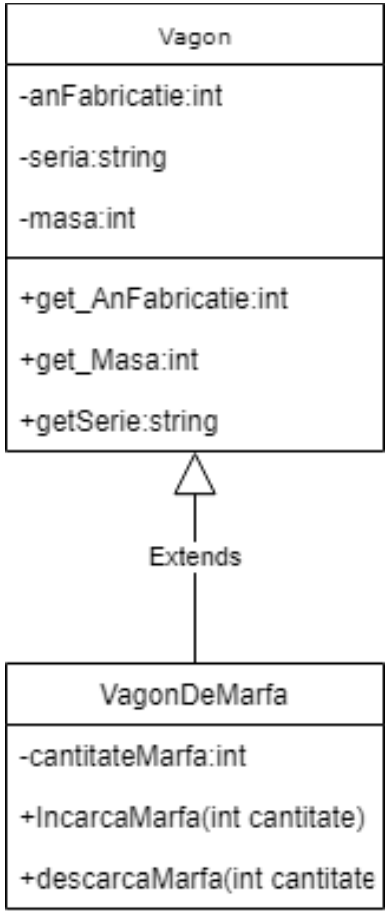


superclasa

subclasa

Moștenirea - inheritance

- Clasa "*fiu*"
  - sau *derivata, subclasa*
  - *Extinde* tipul clasei parinte
  - ***Este*** de tipul clasa parinte
- Campurile *private* ale clasei *parinte*
  - nu sunt **accesibili** in clasa *derivate*
  - **Exista** in continuare ca membri ai clasei *parinte*
- Campurile public ale clasei parinte
  - Sunt membri accesibili in clasa parinte
- Campurile specific (care extend) clasei derivate
  - Nu sunt membri ai clasei parinte
  - Clasa *parinte* – **agnostica** fata de clasa *derivata*



Exemplu

```
class Vagon
{
    3 references
    public string Seria { get; set; } = string.Empty;
    3 references
    public int Masa { get; set; } = 0;
    2 references
    public void Tipareste()
    {
        Console.WriteLine($"Vagonul {Seria} are masa {Masa}");
    }
}
2 references
class VagonDeMarfa : Vagon
{
    private int cantitateMarfa;
    1 reference
    public void IncarcaMarfa(int cantitate)
    {
        cantitateMarfa += cantitate;
    }
    1 reference
    public int DescarcaMarfa()
    {
        int result = cantitateMarfa;
        cantitateMarfa = 0;
        Console.WriteLine($"Am descarcat {cantitateMarfa} tone de marfa");
        return result;
    }
}
```

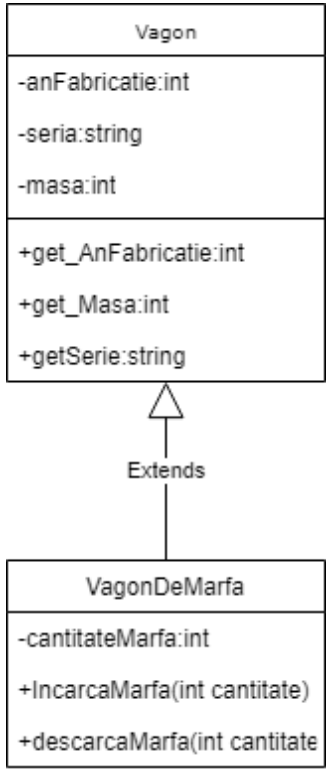
```
static void Main(string[] args)
{
    Vagon vagon = new Vagon();
    vagon.Masa = 123;
    vagon.Seria = "TM11TTL";
    vagon.Tipareste();

    VagonDeMarfa marfar = new VagonDeMarfa();
    marfar.Masa = 444;
    marfar.Seria = "TM44ADP";

    marfar.Tipareste();
    marfar.IncarcaMarfa(50);
    marfar.DescarcaMarfa();
}
```

- Clasa copil
  - Expune toate proprietatile publice ale clasei parinte
  - Poate adauga functionalitati in plus

Tipuri de mostenire

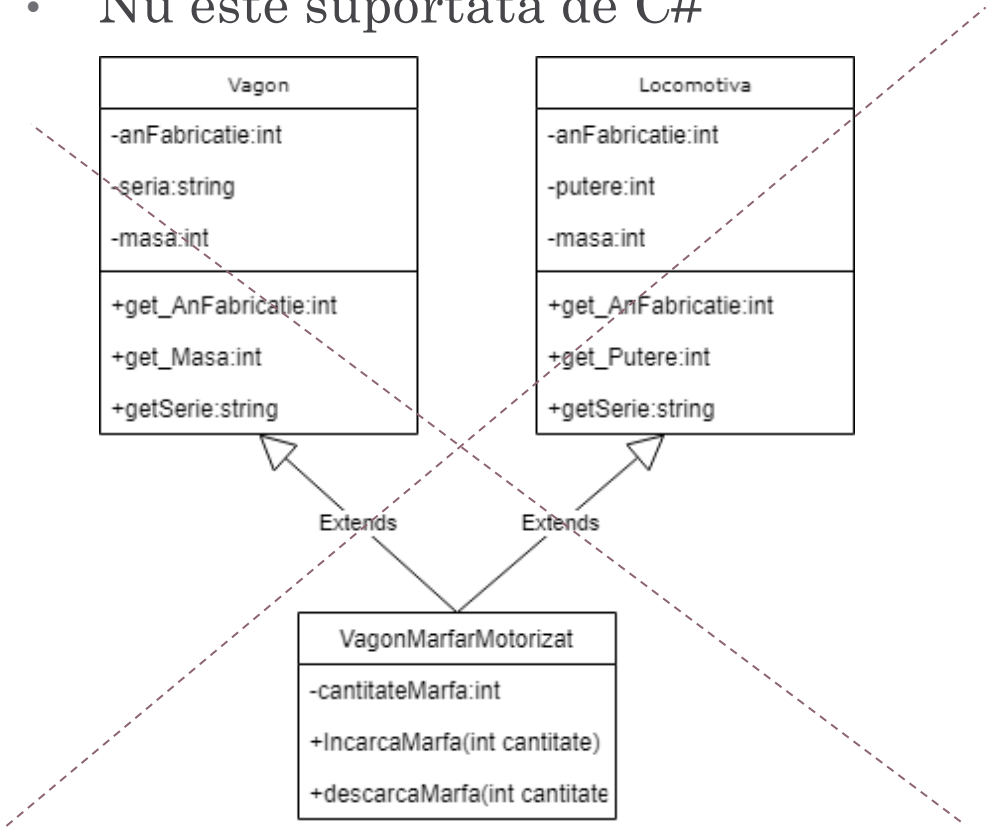
Mostenire simpla

- Suportata in C#



Mostenire multipla

- Nu este suportata de C#

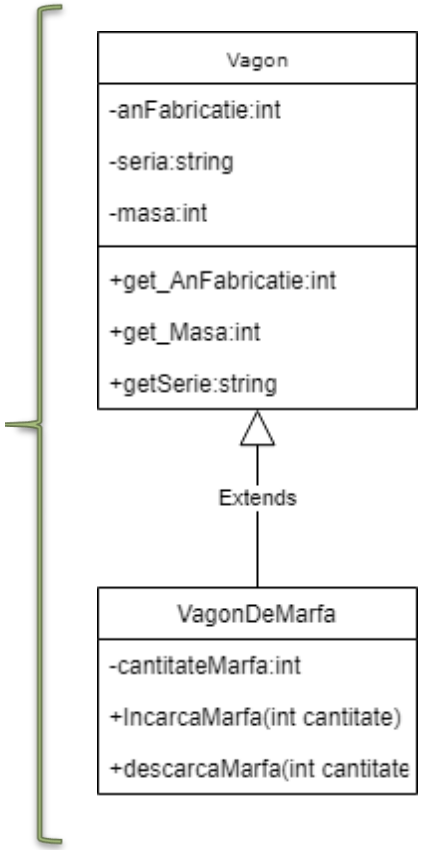


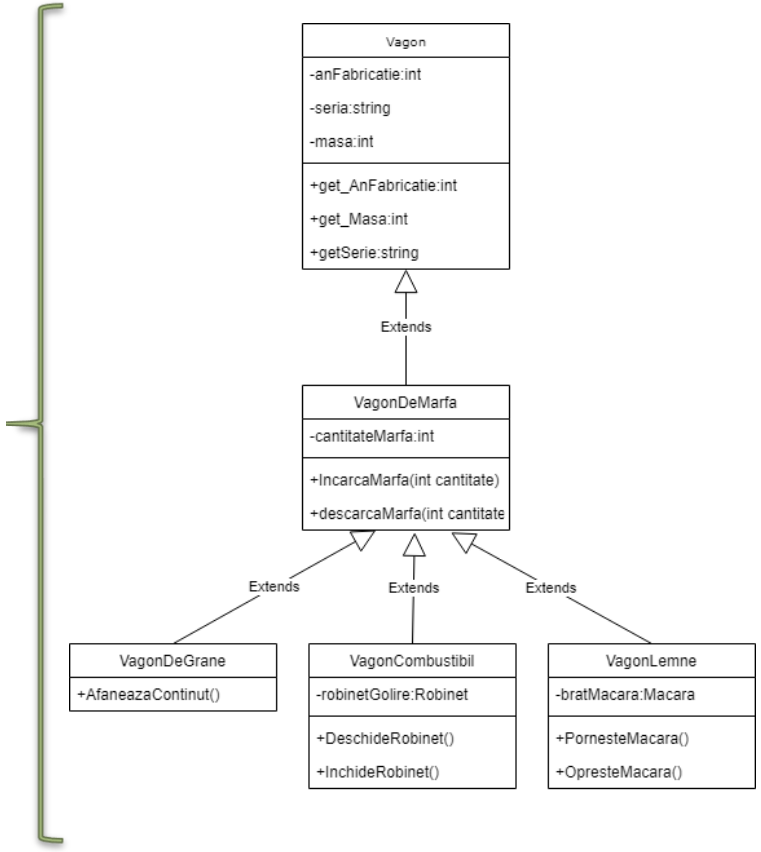
Mostenire – niveluri

Single-level inheritance
- Un singur nivel de mostenire

Multi-level inheritance
- 2 nivele

Class hierarchy
(Ierarhie de clase)





Inheritance - base

```
public int DescarcaMarfa()
{
    int result = cantitateMarfa;
    cantitateMarfa = 0;
    Console.WriteLine($"Am descarcat {cantitateMarfa} tone de marf din vagonul cu seria {base.Seria}");
    return result;
}
```

- *base*
  - referinta spre clasa parinte de nivel imediat precedent

Inheritance – protected

- Protected – access modifier
  - Membrii marcate cu protected sunt
    - Inaccesibili in exteriorul clasei si a ierarhiei de clase
    - Vizibili in interiourul clasei precum si in toate subclasele care o mostenesc, indiferent de numarul de nivele de mostenire intre ele

```
class Vagon
{
    private string serie="";
    private int masa=2400;
    protected TipVagon tipVagon;
    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;
        base.tipVagon = TipVagon.Marfa
    }
}

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

is, as

is, as

- is – in contextul mostenirii
  - operator
  - testeaza daca un obiect poate fi cast-uit intr-un tip
    1. Obiectul este de tipul cerut
       camion is Camion
    2. Obiectul este un subtip al tipului
       camion is Autoturism
- as
  - Converise specifica reference types
  - Nu arunca exceptii (safe)
  - Returneaza null daca nu se realizeaza cu success
  - Permite un null test ulterior
- (cast) – arunca exceptie in caz de eroare

```
Parinte parinte = new Parinte();
Copil copil = new Copil();

if (parinte is Parinte)
{
    Console.WriteLine("parinte este un parinte");
}
else
{
    Console.WriteLine("parinte nu este un parinte");
}
if (copil is Parinte)
{
    Console.WriteLine("copilul este un parinte");
}
else
{
    Console.WriteLine("copilul nu este un parinte");

}
if (parinte is Copil)
{
    Console.WriteLine("parinte este un copil");
}
else
{
    Console.WriteLine("parinte nu este un copil");
}
```

Ce va afisa?

is, as

- is – in contextul mostenirii
  - operator
  - testeaza daca un obiect poate fi cast-uit intr-un tip
    1. Obiectul este de tipul cerut
       camion is Camion
    2. Obiectul este un subtip al tipului
       camion is Autoturism
- as
  - Converise specifica reference types
  - Nu arunca exceptii (safe)
  - Returneaza null daca nu se realizeaza cu success
  - Permite un null test ulterior
- (cast) – arunca exceptie in caz de eroare

```
Parinte parinte = new Parinte();
Copil copil = new Copil();

if (parinte is Parinte)
{
    Console.WriteLine("parinte este un parinte");
}
else
{
    Console.WriteLine("parinte nu este un parinte");
}
if (copil is Parinte)
{
    Console.WriteLine("copilul este un parinte");
}
else
{
    Console.WriteLine("copilul nu este un parinte");

}
if (parinte is Copil)
{
    Console.WriteLine("parinte este un copil");
}
else
{
    Console.WriteLine("parinte nu este un copil");
}
```

is, as

- O subclasa poate fi persistata intr-o referinta de tipul unei clasei parinte, de pe orice nivel superiori
- Conversia – cu *as*
- *Utilizare*
  - Persistam mai multe obiecte din ierarhie intr-un obiect/lista de tipul parintelui
  - Pentru a avea acces la metodele unei clase- copil
    - Is/as

```
Parinte parinte = new Parinte();
Parinte copil = new Copil();

Copil asCopil1 = copil as Copil;
if (asCopil1 != null)
{
    Console.WriteLine("copil->copil != null");
}
else
{
    Console.WriteLine("copil->copil == null");
}

Copil asCopil2 = parinte as Copil;
if (asCopil2 != null)
{
    Console.WriteLine("parinte->copil != null");
}
else
{
    Console.WriteLine("parinte->copil == null");
}
```

is, as

```
class Parinte
{
    1 reference
    public void Tipareste()
    {
        Console.WriteLine("buna, sunt parinte");
    }
}
6 references
class Copil : Parinte
{
    0 references
    public void PrezintaParintele()
    {
        Console.WriteLine("Sa va prezint parintele");
        base.Tipareste();
    }
}
```

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

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

    if (copil is Copil)
    {
        Copil asCopil = copil as Copil;
        asCopil.PrezintaParintele();
        (copil as Copil).PrezintaParintele();
    }
}
```

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

Va multumesc!