



## Modulul 9

### Noțiuni avansate de POO

Acest modul oferă informații avansate despre concepte de programare orientată pe obiect. Vor fi explicate conceptele de:

- **moștenire** : derivarea claselor C# din clase existente, modul de folosire a cuvintelor cheie **virtual**, **override** și **new**.

- **clase abstracte**

- **interfețe**

- **poliforfism**



## Overview

- ▶ Derivarea claselor
- ▶ Folosirea claselor abstracte
- ▶ Folosirea interfețelor
- ▶ Folosirea polimorfismului



2

- La finalul acestui modul, veți putea să:
  - Derivați clase și să implementați interfețe
  - Creați și să folosiți clasele abstracte
  - Creați și să folosiți interfețe
  - Implementați conceptul de polimorfism



## Derivarea claselor - Introducere

- Capacitatea de a crea o clasă de bază cu proprietăți și metode care pot fi folosite în clase extinse din aceasta.
- Exemplu și sintaxă:

```
class Human
{
    public string name;
    public int age;
}

class Student: Human
{
    public string universityName;
}
```

3

- Derivarea unei clase reprezintă capacitatea de a crea o clasă de bază cu proprietăți și metode care pot fi folosite în clase extinse din aceasta.
- Sintaxa pentru moștenirea unei clase de bază este următoarea:

```
class Human
{
    public string name;
    public int age;
}

class Student: Human
{
    public string universityName;
}
```

- În exemplul de mai sus, clasa de bază este reprezentată de „**Human**”, pe când clasa derivată din aceasta se numește „**Student**”. Pentru a specifica faptul că o clasă este derivată dintr-o clasă de bază se folosește operatorul „:”, urmat de numele clasei de bază.


 itacad  
 you@technology

 Microsoft  
 .NET

## Derivarea claselor - Moștenire

```
class Human
{
    public string name;
    public int age;
}

class Student: Human
{
    public string universityName;
}
```

- ▶ O clasă derivată moștenește elementele clasei de bază (câmpuri și metode)
- ▶ Relația: clasa derivată **este un/o** clasă de bază.

4

➤ O clasă derivată moștenește majoritatea elementelor clasei de bază. Astfel, în exemplul prezentat, clasa **Student** moștenește de la clasa **Human** câmpurile **name** și **age**. Clasa derivată poate declara oricâte alte elemente în plus, pe lângă cele moștenite, de exemplu **universityName**, care reprezintă numele universității din care face parte. Atenție, chiar dacă o clasă derivată moștenește toate elementele clasei de bază, ea nu le poate folosi pe cele declarate **private** în clasa de bază.

➤ Relația dintre o clasă de bază și o clasă derivată din aceasta este „**is-a**” (este un/o). Se poate observa această relație între clasele prezentate deoarece un student **este un** om. Astfel, câmpurile ca nume și vârstă prezente în clasa **Human** vor fi moștenite în clasa **Student**, urmând ca cea din urmă să adauge câmpuri care nu sunt specifice neapărat și clasei de bază (**numele universității în care învață**).


## Derivarea claselor – Modificatori de acces

- O clasă derivată nu poate fi mai accesibilă decât clasa de bază
- Membrii marcați ca **protected** sunt implicit **protected** și în clasa derivată

```
class Token
{
    ...
    protected string name;
}
class CommentToken: Token
{
    ...
    public string Name()
    {
        return name;
    }
}
```



```
Class Outside
{
    void Fails(Token t)
    {
        Console.Write(t.name);
    }
}
```



5

► O clasă derivată nu poate fi mai accesibilă decât clasa de bază. Astfel, dacă o clasă de bază este declarată ca fiind **protected**, clasa derivată din aceasta nu poate fi declarată ca fiind **public**.

► Modificatorii de acces folosiți în clasa de bază au efect și în clasa derivată. De exemplu, membrii marcați ca **protected** în clasa de bază sunt implicit **protected** și în clasa derivată.

► Se poate observa că membrul **name** din clasa Token poate fi accesat din clasa derivată (deoarece a fost moștenit), însă datorită modificatorului de acces **protected** (care permite accesul la câmp din clasa în care este declarat și din clasele derivate din aceasta, dar nu și din clasele externe), acesta nu poate fi accesat dintr-o clasă externă. Atenție, un membru declarat în clasa de bază ca fiind **private** nu poate fi accesat din clasele derivate din aceasta, deși acesta este moștenit.


itacad  
you@technology

Microsoft  
.NET

## Derivarea claselor – Constructorii clasei de bază

- ▶ Se folosește cuvântul cheie **base**

```
class Token
{
    protected Token(string name) {...}
}

class CommentToken: Token
{
    public CommentToken(string name) : base(name) {...}
    ...
}
```

- ▶ Un constructor privat nu poate fi accesat de către o clasă derivată

6

Pentru a apela constructorul clasei de bază din constructorul clasei derivate, se folosește cuvântul cheie **base**. Sintaxa este următoarea:

**public CommentToken(string name): base(name) {...}**

unde CommentToken(string name) este constructorul clasei derivate.

### Declararea constructorilor

Dacă clasa derivată nu apelează explicit un constructor, compilatorul C# va folosi, în mod implicit, constructorul de inițializare de forma “: base()”

Acest comportament implicit este util pentru că:

- o clasă ce nu extinde explicit nici o clasă extinde, implicit, clasa **System.Object**, care conține un constructor public, fără parametri.
- dacă o clasă nu conține un constructor, compilatorul va oferi automat un constructor fără parametri, numit constructor implicit.

Dacă o clasă oferă un constructor propriu, compilatorul nu va mai crea constructorul implicit. Totuși, dacă constructorul specificat nu identifică nici un constructor din clasa de bază, compilatorul va genera o eroare:

```
class Token
{
    protected Token(string name) { ... }
}
class CommentToken: Token
{
    public CommentToken(string name) { ... } // Error here
}
```


 itacad  
 you@technology

 Microsoft  
 .NET

## Derivarea claselor – Constructorii clasei de bază

- ▶ Se folosește cuvântul cheie **base**

```
class Token
{
    protected Token(string name) {...}
}

class CommentToken: Token
{
    public CommentToken(string name) : base(name) {...}
    ...
}
```

- ▶ Un constructor privat nu poate fi accesat de către o clasă derivată

7

Această eroare apare pentru că, la declararea constructorului din **CommentToken**, se inițializează implicit constructorul “: base()”, iar acesta nu există în clasa de bază și nici nu este generat implicit din cauza declarării explicite a unui constructor, ce primește un parametru de tip **string**. Această eroare se poate repara folosind codul de pe slide.


### Reguli de acces la constructori

Regulile de acces pentru un constructor derivat ce apelează un constructor al clasei de bază sunt aceleași ca și pentru metodele normale. De exemplu, dacă constructorul clasei de bază este privat, clasa derivată nu îl va putea accesa.

```
class NonDerivable
{
    private NonDerivable( ) { ... }
}


class Impossible: NonDerivable
{
    public Impossible( ) { ... } // Compile-time error
}
```

În acest caz, nu se poate ca o clasă derivată să apeleze constructorul clasei de bază

 itacad  
you@technology

## Derivarea claselor - Implementarea metodelor

- ▶ Metode virtuale
- ▶ Metode suprascrise
- ▶ Folosirea cuvântului **new** pentru a ascunde metode
- ▶ Exemplu de implementare a metodelor

 Microsoft  
.NET

8

Metodele implementate într-o clasă de bază pot fi redefinite într-o clasă derivată dacă acestea au fost create pentru a permite suprascrierea.

Veți învăța să:

- declarați metode **virtual**
- declarați metode **override**
- ascundeți metode




 itacad  
 you@technology

 Microsoft  
 .NET

## Derivarea claselor - Definirea metodelor virtuale

### ► Declararea unei metode virtuale

```
class Token
{
    public int LineNumber()
    {
        ...
    }
    public virtual string Name()
    {
        ...
    }
}
```

- Metodele virtuale sunt polimorfice
- Metodele virtuale nu pot fi declarate ca statice
- Metodele virtuale nu pot fi declarate ca private

9

O metodă ce conține în definiție cuvântul cheie **virtual** specifică o implementare a unei metode ce poate fi suprascrisă polimorfic într-o clasă derivată.

O metodă ce nu a fost declarată ca **virtual** specifică *singura* implementare a respectivei metode. Nu veți putea suprascrie polimorfic o metodă non-virtuală într-o clasă derivată. Acest lucru înseamnă că nu veți putea declara o metodă în clasa derivată care să aibă aceeași semnătură ca cea din clasa de bază pentru a schimba comportamentul acesteia.

### Sintaxa

Pentru a declara o metodă virtuală, folosiți cuvântul cheie **virtual**. Sintaxa pentru acest cuvânt cheie este prezentată în slide.

Când se declară o metodă virtuală, este obligatoriu ca aceasta să conțină și un corp. Dacă nu se oferă implementarea, compilatorul va genera o eroare.

Pentru a folosi metode virtuale eficient, trebuie să înțelegeți următoarele:

- nu puteți declara metode virtuale ca statice - deoarece metodele statice sunt metode ale clasei și nu pot fi polimorfice (polimorfismul se aplică obiectelor, nu claselor)
- nu puteți declara metode virtuale ca private- deoarece ele nu pot fi accesate pentru a fi suprascrise într-o clasă derivată.



## Derivarea claselor - Suprascrierea metodelor

- ▶ Se pot suprascrive metode moștenite ce sunt marcate ca **virtual** sau ca **override**

```
class Token
{
    public virtual string Name() {...}
}
class CommentToken: Token
{
    public override string Name() {...}
}
```

- ▶ O metodă override nu poate să fie declarată explicit ca virtuală

10

O metodă suprascrisă specifică o altă implementare a unei metode virtuale. Metodele declarate ca virtuale în clasa de bază pot fi suprascrise polimorfic în clasele derivate.

### Sintaxa

Se folosește cuvântul cheie **override** pentru a defini o metodă suprascrisă, ca în exemplul următor:

```
class Token
{ ...
    public virtual string Name( ) { ... }
}

class CommentToken: Token
{ ...
    public override string Name( ) { ... }
}
```

## Derivarea claselor - Suprascrierea metodelor

- ▶ Se pot suprascrive metode moștenite ce sunt marcate ca **virtual** sau ca **override**

```
class Token
{
    public virtual string Name() {...}
}
class CommentToken: Token
{
    public override string Name() {...}
}
```

- ▶ O metodă override nu poate să fie declarată explicit ca virtuală

11

La fel ca și pentru metodele virtuale, trebuie oferită o implementare a metodei, altfel compilatorul va genera o eroare:

```
class Token
{
    public virtual string Name( ) { ... }
}

class CommentToken: Token
{
    public override string Name( ); // Eroare la compilare
}
```

Cuvântul cheie **override** îi spune compilatorului ca acea metodă există deja în clasa de bază și că vrei să îi schimbi comportamentul (corpul metodei). Dacă vei greși semnătura metodei compilatorul te va alerta că metoda pe care vrei să o suprascreești nu există în clasa de bază.



## Derivarea claselor - Folosirea metodelor suprascrise

```
class Token
{
    public int LineNumber() {...}
    public virtual string Name() {...}
}

class CommentToken: Token
{
    public override int LineNumber() {...}
    public override string Name() {...}
}
```

- Se pot suprascrie doar metode virtuale indentice
- O metodă override nu poate să fie declarată ca statică sau privată.

12

Următoarele reguli trebuie știute în momentul în care se lucrează cu metode suprascrise:

- se pot suprascrie doar metode identice ce au fost definite ca virtuale
- metoda override trebuie să fie identică metodei virtuale
- se poate suprascrie o metodă override
- nu se poate declara o metodă override ca virtuală
- nu se poate declara o metodă override ca privată sau statică

**Se pot suprascrie doar metode identice ce au fost definite ca virtuale.**

După cum se observă în codul din slide, metoda `LineNumber` din clasa derivată va genera eroare de compilare deoarece metoda de bază nu este marcată ca virtuală.

**Metoda override trebuie să fie identică metodei virtuale**

O metodă **override** trebuie să fie identică în toate aspectele metodei virtuale pe care o suprascrie. Trebuie să aibă același nivel de acces, același tip returnat, același nume, aceiași parametri.

De exemplu, suprascrierea din exemplul următor eșuează deoarece nivelul de acces este diferit (`protected` spre deosebire de `public`), valoarea întoarsă este diferită (`string` spre deosebire de `void`) și parametrii sunt, de asemenea, diferiți.

```
class Token
{
    protected virtual string Name() { ... }
}
class CommentToken: Token
{
    public override void Name(int i) { ... } // Errors
}
```

**Se poate suprascrie o metodă override**

O metodă **override** este implicit și **virtual**, așa că se poate suprascrie.

```
class Token
{
    public virtual string Name() { ... }
}
class CommentToken: Token
{
    public override string Name() { ... }
}
class OneLineCommentToken: CommentToken
{
    public override string Name() { ... } // Okay
}
```

Totuși, nu se poate specifica explicit că o metodă **override** este **virtual**.

```
class Token
{
    public virtual string Name() { ... }
}
class CommentToken: Token
{
    public virtual override string Name() { ... } // Error
}
```

**Nu se poate declara o metodă override ca statică sau privată**

Nici metodele statice, nici cele private nu sunt polimorfice. Din acest motiv, metodele virtuale și cele override nu pot fi declarate statice sau private.



## Derivarea claselor - Folosirea metodelor suprascrise

```
class Token
{
    public int LineNumber() {...}
    public virtual string Name() {...}
}

class CommentToken: Token
{
    public override int LineNumber() {...}
    public override string Name() {...}
}
```

- ▶ Se pot suprascrie doar metode virtuale indente
- ▶ O metodă override nu poate să fie declarată ca statică sau privată.

13

### Se poate suprascrie o metodă override

O metodă **override** este implicit și **virtual**, așa că se poate suprascrie.

```
class Token
{
    public virtual string Name( ) { ... }
}
class CommentToken: Token
{
    public override string Name( ) { ... }
}
class OneLineCommentToken: CommentToken
{
    public override string Name( ) { ... } // Okay
}
```

Totuși, nu se poate specifica explicit că o metodă **override** este **virtual**.

```
class Token
{
    public virtual string Name( ) { ... }
}
class CommentToken: Token
{
    public virtual override string Name( ) { ... } // Error
}
```

### Nu se poate declara o metodă override ca statică sau privată

Nici metodele statice, nici cele private nu sunt polimorfice. Din acest motiv, metodele virtuale și cele override nu pot fi declarate statice sau private.


itacad  
you@technology

Microsoft  
.NET

## Derivarea claselor - Întrebare

### Întrebare fulger:

Care este diferența dintre supraîncărcarea și suprascrierea metodelor?

### Răspuns:

**Supraîncărcarea:** metode cu același nume dar cu semnături diferite

**Suprascrierea:** metodă în clasa derivată cu același nume și cu aceeași semnătură ca cea din clasa de bază

14

Întrebare fulger: Care este diferența dintre supraîncărcare și suprascriere?

Răspuns:

**Supraîncărcarea** metodelor este procesul prin care se definesc multiple metode cu același nume dar cu semnături diferite .

(Obs: Semnătura unei metode este combinația dintre nume și lista de parametri).

**Suprascrierea** este procesul prin care într-o clasă derivată declarăm o metodă cu aceeași semnătură ca metoda din clasa de bază și în interiorul căreia definim un nou comportament.



## Folosirea **new** pentru a ascunde o metodă

- ▶ Se poate ascunde o metodă moștenită introducând o nouă metodă în clasă
- ▶ Metoda moștenită din clasa de bază va fi înlocuită de o metodă complet diferită.

```
class Token
{
    ...
    public int LineNumber( ) { ... }
}

class CommentToken: Token
{
    ...
    new public int LineNumber( ) { ... }
}
```

15

Se poate ascunde o metodă identică moștenită de la o clasă introducând o nouă metodă între membrii clasei. Vechea metodă ce a fost moștenită de către clasa derivată este înlocuită de către o metodă complet diferită.

### Sintaxa

Se folosește cuvântul cheie **new** pentru a introduce o metodă ce ascunde o metodă moștenită:

```
class Token
{ ...
    public int LineNumber( ) { ... }
}

class CommentToken: Token
{ ...
    new public int LineNumber( ) { ... }
}
```



## Folosirea **new** pentru a ascunde o metodă

- ▶ Se poate folosi pentru ascunderea atât a metodelor virtuale, cât și non-virtuale

```
class Token
{
    ...
    public int LineNumber( ) { ... }
    public virtual string Name( ) { ... }
}
class CommentToken: Token
{
    ...
    new public int LineNumber( ) { ... }
    public override string Name( ) { ... }
}
```

- ▶ Ascunde metode ce au semnături identice

16

Folosind cuvântul cheie **new**, se pot face următoarele operații:

- ascunde atât metode virtuale, cât și non-virtuale
- rezolva conflicte de nume în cod
- ascunde metode cu semnături identice

### Ascunderea metodelor virtuale și non-virtuale

Folosirea lui **new** pentru a ascunde metode are implicații doar dacă se folosește polimorfismul. De exemplu, în codul din slide, **CommentToken.LineNumber** este o metodă nouă. Nu are nici o legătură cu metoda **Token.LineNumber**. Chiar dacă **Token.LineNumber** ar fi fost o metodă virtuală, **CommentToken.LineNumber**, în continuare, nu ar fi fost asociată cu această metodă.

În acest exemplu, **CommentToken.LineNumber** nu este virtuală. Asta înseamnă că orice derivare a clasei **CommentToken** nu va putea suprascrie metoda **LineNumber**. Totuși, noua metodă **CommentToken.LineNumber** ar putea fi declarată virtuală, în acest caz derivările ulterioare ar putea suprascrie această metodă:

```
class CommentToken: Token
{
    ...
    new public virtual int LineNumber( ) { ... }
}
class OneLineCommentToken: CommentToken
{
    public override int LineNumber( ) { ... }
}
```



## Folosirea **new** pentru a ascunde o metodă

- ▶ Se poate folosi pentru ascunderea atât a metodelor virtuale, cât și non-virtuale

```
class Token
{
    ...
    public int LineNumber( ) { ... }
    public virtual string Name( ) { ... }
}
class CommentToken: Token
{
    ...
    new public int LineNumber( ) { ... }
    public override string Name( ) { ... }
}
```

- ▶ Ascunde metode ce au semnături identice

17

### Rezolvarea conflictelor de nume în cod

Conflictele de nume generează, cel mai des, mesaje de avertizare la compilare. De exemplu, considerând următorul cod:

```
class Token
{
    public virtual int LineNumber( ) { ... }
}
class CommentToken: Token
{
    public int LineNumber( ) { ... }
}
```

La compilarea acestui cod, se va obține un mesaj de avertizare ce anunță că **CommentToken.LineNumber** ascunde **Token.LineNumber**. Acest avertisment anunță conflictul de nume. Aveți 3 moduri în care puteți să rezolvați acest avertisment:

1. Adăugați cuvântul cheie **override** la declararea **CommentToken.LineNumber**
2. Marcați explicit faptul că **CommentToken.LineNumber** va ascunde metoda **Token.LineNumber**. Pentru a face acest lucru, folosiți cuvântul cheie **new**.
3. Schimbați numele metodei.

### Ascunderea metodelor ce au semnături identice

Modificatorul **new** este necesar când, la derivarea unei clase, se dorește ascunderea unei metode a clasei de bază ce are semnătură identică. În exemplul următor, compilatorul avertizează că modificatorul **new** nu este necesar, întrucât metoda clasei de bază are o semnătură diferită de cea a clasei derivate:

```
class Token
{
    public int LineNumber(short s) { ... }
}
class CommentToken: Token
{
    new public int LineNumber(int i) { ... } // Warning
}
```



## Identificați greșelile

```
class Base
{
    public void Alpha( ) { ... }
    public virtual void Beta( ) { ... }
    public virtual void Gamma(int i) { ... }
    public virtual void Delta( ) { ... }
    private virtual void Epsilon( ) { ... }
}
class Derived: Base
{
    public override void Alpha( ) { ... }
    protected override void Beta( ) { ... }
    public override void Gamma(double d) { ... }
    public override int Delta( ) { ... }
}
```

18

- 1) Clasa de bază declară o metodă virtuală și privată. Acest lucru nu se poate face
- 2) Clasa derivată declară metoda **Alpha** folosind modificatorul **override**, deși clasa de bază nu este marcată ca virtuală
- 3) Clasa derivată declară o metodă **protected Beta** folosind modificatorul **override**. Totuși, metoda clasei de bază este **public**, modificatorul de acces neputând fi modificat
- 4) Clasa derivată declară o metodă numită **Gamma** folosind modificatorul **override**. Metoda din clasa derivată ce încearcă suprascrierea acestei metode primește un alt set de parametri.
- 5) Clasa derivată declară o metodă publică **Delta** folosind modificatorul **override**. Există o diferență de tip returnat față metoda definită în clasa de bază.


itacad  
you@technology

Microsoft  
.NET

## Folosirea claselor sigilate

- ▶ Clasele sigilate nu pot fi moștenite
- ▶ Clasele sigilate sunt folosite pentru optimizarea operațiilor la run-time
- ▶ Multe dintre clasele din platforma .NET sunt sigilate
- ▶ Se folosește cuvântul cheie **sealed**

```
namespace System
{
    public sealed class String
    {
        ...
    }
}
namespace Mine
{
    class FancyString: String { ... }
}
```



19

Crearea unei ierarhii de moștenire flexibilă nu este ușoară. Multe clase sunt clase de sine stătătoare care nu sunt gândite pentru a fi derivate. Totuși, din punct de vedere al sintaxei, derivarea unei clase este foarte ușoară și implică folosirea doar a unor cuvinte cheie. Ușurința derivării permite posibilitatea periculoasă ca programatorii să deriveze clase ce nu au fost create pentru a fi derivate.

Pentru a înlătura această problemă și pentru a face mai clar scopul unor anumite clase, există posibilitatea de a declara clase **sealed** care nu permit să fie moștenite.

### Sintaxa

O clasă se sigilează folosind cuvântul cheie **sealed**. Sintaxa pentru o astfel de clasă este:

```
namespace System
{
    public sealed class String
    {
        ...
    }
}
```

### Optimizarea operațiilor pentru run-time

Modificatorul **sealed** activează anumite optimizări pentru run-time. În particular, deoarece o clasă **sealed** nu va fi derivată, este posibil să se transforme toate funcțiile membre din **virtual** în non-virtual.



## Clase abstracte

### ► Sintaxa:

```
abstract class Token
{
    public virtual string Name( ) { ... }
    public abstract int Length( );
}
class CommentToken: Token
{
    public override string Name( ) { ... }
    public override int Length( ) { ... }
}
```

- Metode abstracte se pot declara doar în clase abstracte
- Metodele abstracte nu au implementare

20

Se declară o metodă abstractă adăugând modificatorul **abstract** la declararea metodei.

Numai clase abstracte pot declara metode abstracte.

### Metodele abstracte nu conțin implementări

```
abstract class Token
{
    public abstract string Name( ) { ... } // Eroare la compilare
}
```


itacad  
you@technology

Microsoft  
.NET

## Folosirea metodelor abstracte

- ▶ Metodele abstracte sunt virtuale
- ▶ Metodele **override** pot suprascrie metode abstracte în clase derivate
- ▶ Metodele abstracte pot suprascrie metode ale clasei de bază declarate ca virtual sau override

21

### Metodele abstracte sunt virtuale

Metodele sunt considerate implicit virtuale dar nu pot fi și explicit marcate ca virtual, după cum se arată în următorul cod:

```
abstract class Token
{
    public virtual abstract string Name( ); // Compile-time error
}
```

### Metodele override pot suprascrie metode abstracte în clase derivate

```
class CommentToken: Token
{
    public override string Name( ) {...}
}
```

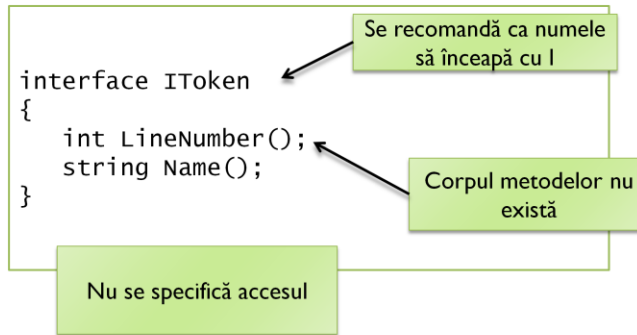
### Metodele abstracte pot suprascrie metode ale clasei de bază declarate ca virtual sau override

În cazul în care o clasă abstractă moștenește o clasă concretă sau o altă clasă abstractă, poate suprascrie membri ai acesteia ce au fost declarați folosind **virtual** sau **override** folosind metode abstracte

```
class Token
{
    public virtual string Name( ) { ... }
}
abstract class Force: Token
{
    public abstract override string Name( );
}
```

## Interfețe - declarare

- Folosiți cuvântul cheie **interface** pentru a declara interfețe



22

O interfață este de fapt o clasă fără cod. O interfață se declară similar modului în care se declară o clasă. Pentru a declara o interfață C#, trebuie folosit cuvântul cheie **interface** în locul cuvântului **class**. Sintaxa este explicată în slide.

**Obs.** Se recomandă ca numele interfețelor să înceapă cu litera 'I' de la „interface”.

### Proprietăți ale interfețelor

Următoarele sunt două proprietăți importante ale interfețelor:

#### Metodele interfețelor sunt implicit publice

Nu se permite specificarea explicită a unor modificatori de acces, nici măcar public

```
interface IToken
{
    public int LineNumber( ); // Eroare la compilare
}
```

#### Metodele interfețelor nu conțin implementare

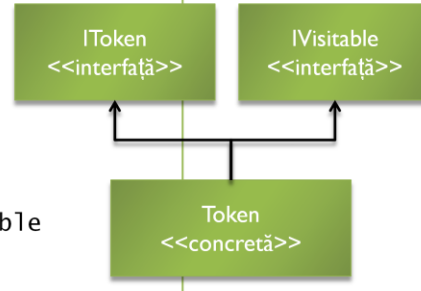
Nu se poate oferi implementare pentru nici o metodă dintr-o interfață.

```
interface IToken
{
    int LineNumber( ) { ... } // Eroare la compilare
}
```

## Interfețe - Implementarea mai multor interfețe

- O clasă poate să implementeze mai multe interfețe

```
interface IToken
{
    string Name( );
}
interface IVisitable
{
    void Accept(IVisitor v);
}
class Token: IToken, IVisitable
{ ...
}
```



- O clasă poate să fie mai accesibilă decât interfața de bază
- O clasă concretă trebuie să implementeze toate metodele din interfață

23

Deși C# permite moștenirea unei singure clase concrete sau abstracte, este permisă implementarea mai multor interfețe într-o clasă.

O interfață nu poate extinde nici o clasă abstractă sau concretă, dar poate extinde oricâte interfețe:

```
interface IToken { ... }
interface IVisitable { ... }
interface IVisitableToken: IVisitable, IToken { ... }
class Token: IVisitableToken { ... }
```

### Accesibilitate

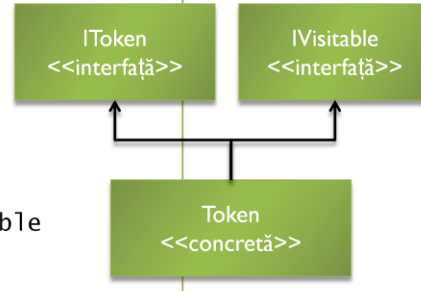
O clasă poate fi mai accesibilă decât interfața de bază. De exemplu, se poate declara o clasă publică ce implementează o interfață privată:

```
class Example
{
    private interface INested { }
    public class Nested: INested { } // Okay
}
```

## Interfețe - Implementarea mai multor interfețe

- O clasă poate să implementeze mai multe interfețe

```
interface IToken
{
    string Name( );
}
interface IVisitable
{
    void Accept(IVisitor v);
}
class Token: IToken, IVisitable
{ ...
}
```



- O clasă poate să fie mai accesibilă decât interfața de bază
- O clasă concretă trebuie să implementeze toate metodele din interfață

24

Totuși, o interfață nu poate fi mai accesibilă decât oricare dintre interfețele de bază.

```
class Example
{
    private interface INested { }
    public interface IAlsoNested: INested { } // eroare la compilare
}
```

### Membrii interfețelor

O clasă concretă trebuie să implementeze toate metodele interfețelor pe care le extinde, chiar dacă acestea sunt moștenite direct sau indirect.



## Interfețe - Implementarea metodelor interfețelor

- ▶ Metoda ce implementează trebuie să fie identică cu cea din interfață
- ▶ Metoda ce implementează poate să fie virtuală sau nu

```
class Token: IToken, IVisitable
{
    public virtual string Name( )
    { ...
    }
    public void Accept(IVisitor v)
    { ...
    }
}
```

25

Când o clasă implementează o interfață, trebuie implementată fiecare metodă declarată în interfață. Această restricție apare deoarece interfețele nu pot defini propriile corpuri ale metodelor.

Metoda din clasa ce implementează o interfață trebuie să fie identică cu metoda interfeței în toate aspectele. Trebuie să aibă același:

- Acces
- Tip de întoarcere
- Nume
- Parametri

Următorul cod respectă toate aceste reguli:

```
interface IToken
{
    string Name( );
}
```



## Interfețe - Implementarea metodelor interfețelor

- ▶ Metoda ce implementează trebuie să fie identică cu cea din interfață
- ▶ Metoda ce implementează poate să fie virtuală sau nu

```
class Token: IToken, IVisitable
{
    public virtual string Name( )
    { ...
    }
    public void Accept(IVisitor v)
    { ...
    }
}
```

26

interface IVisitable

```
{
    void Accept(IVisitor v);
}
```

class Token: IToken, IVisitable

```
{
    public virtual string Name( )
    { ...
    }
    public void Accept(IVisitor v)
    { ...
    }
}
```

**Obs.** Metodele ce implementează pot fi virtuale sau nu.

## Interfețe - Implementarea explicită a metodelor

- ▶ Se folosește numele complet al numelui metodei interfeței

```
class Token: IToken, IVisitable
{
    string IToken.Name( )
    { ...
    }
    void IVisitable.Accept(IVisitor v)
    { ...
    }
}
```

- ▶ Restricții ale implementării explicite
  - ▶ Metodele pot fi accesate doar prin interfață
  - ▶ Metodele nu pot fi declarate ca virtuale
  - ▶ Nu se poate specifica un modificador de acces

27

O cale alternativă pentru a implementa o metodă moștenită de la o interfață este de a specifica explicit implementarea metodei interfeței

### Folosirea numelui complet al metodei interfeței

Aceasta înseamnă specificarea interfeței pe lângă numele metodei. Un exemplu este afișat pe slide. Observați diferențele față de implementarea anterioară

### Restricții ale implementării explicite

#### Metodele pot fi accesate doar prin interfață

Metodele vor putea fi accesate doar folosind cast la tipul interfeței:

```
class Token: IToken, IVisitable
{
    string IToken.Name( )
    {
        ...
    }
    private void Example( )
    {
        Name( ); // Eroare de compilare
        ((IToken)this).Name( ); // Okay
    }
    ...
}
```

## Interfețe - Implementarea explicită a metodelor

- ▶ Se folosește numele complet al numelui metodei interfeței

```
class Token: IToken, IVisitable
{
    string IToken.Name( )
    { ...
    }
    void IVisitable.Accept(IVisitor v)
    { ...
    }
}
```

- ▶ Restricții ale implementării explicite
  - ▶ Metodele pot fi accesate doar prin interfață
  - ▶ Metodele nu pot fi declarate ca virtuale
  - ▶ Nu se poate specifica un modificador de acces

28

### Nu pot fi declarate metode ca virtuale

O clasă ce derivează în continuare această clasă nu va putea să acceseze o implementare a unei metode interfață și nu o va putea suprascrie.

### Nu se poate specifica un modificador de acces

Când se face o astfel de implementare, nu se poate specifica un modificador de acces. Aceste metode sunt publice și nu se poate face o altfel de implementare.

### Nu se poate face acces direct la metode

Următorul cod va genera eroare la compilare:

```
class InOneSensePrivate
{
    void Method(Token t)
    {
        t.Name( ); // Eroare de compilare
    }
}
```

Accesul se poate face numai printr-o variabilă de tipul interfeței:

```
class InAnotherSensePublic
{
    void Method(Token t)
    {
        ((IToken)t).Name( ); // Okay
    }
}
```

## Identificați greșelile

```
interface IToken
{
    string Name( );
    int LineNumber( ) { return 42; }
    string name;
}
class Token
{
    string IToken.Name( ) { ... }
    static void Main( )
    {
        IToken t = new IToken( );
    }
}
```

29

1. Interfața **IToken** declară o metodă **LineNumber** și oferă și corp pentru această metodă
2. Interfața **IToken** conține o variabilă **name**, dar nu poate conține nici un fel de implementare.
3. Clasa **Token** conține o implementare a metodei **IToken.Name()**, dar nu s-a precizat faptul că extinde această interfață.
4. Nu se oferă în clasa **Token** o implementare și pentru **LineNumber()**;
5. Se încearcă în metoda **Main** a clasei **Token** instanțierea unei interfețe. Acest lucru nu este posibil.



## Comparație între clase abstracte și interfețe

- ▶ **Asemănări**
  - ▶ Nu pot fi instanțiate
  - ▶ Nu sunt sigilate (pot fi moștenite)
- ▶ **Diferențe**
  - ▶ Interfețele nu conțin implementări
  - ▶ Interfețele nu pot declara membri care nu sunt publici
  - ▶ Interfețele nu pot extinde tipuri care nu sunt interfețe

30

Atât clasele abstracte cât și interfețele există doar pentru a fi derivate (sau implementate). Totuși, o clasă poate extinde cel mult o clasă abstractă, această restricție neexistând pentru interfețe.

### Asemănări

- Nu pot fi instanțiate
- Nu sunt sigilate (pot fi moștenite)

### Diferențe

- Interfețele nu conțin implementări; Clasele abstracte pot să conțină implementări
- Interfețele nu pot declara membri care nu sunt publici; clasele abstracte pot să conțină membrii privați.
- Interfețele nu pot extinde clase care nu sunt interfețe; clasele abstracte pot extinde clase concrete, alte clase abstracte sau interfețe.



## Identificați greșelile

```

▶ class First
{
    public abstract void Method();
}
    
```

```

▶ abstract class Second
{
    public abstract void Method() {}
}
    
```

```

▶ interface IThird
{
    void Method();
}
abstract class Third: Ithird
{
}
    
```

31

1. Metode abstracte se pot declara doar în clase abstracte
2. O metodă abstractă nu poate avea implementare
3. O clasă abstractă ce implementează o interfață va trebui fie să implementeze toți membrii interfeței, fie să definească metode abstracte pentru membrii ce nu au fost implementați.



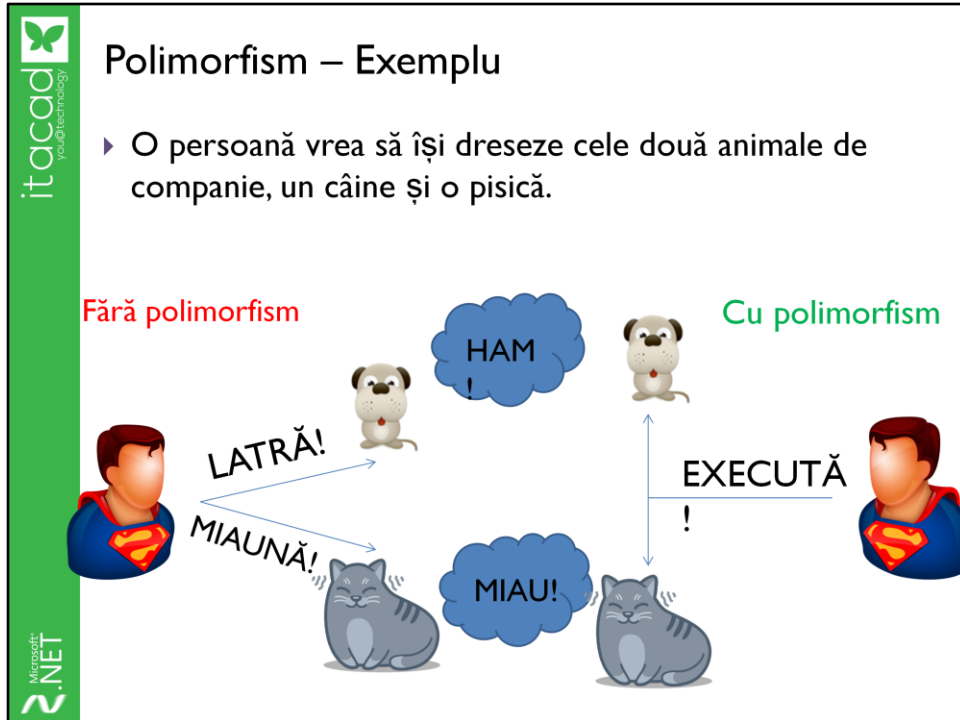
## Polimorfism – Definiție

- Capacitatea a cel puțin două clase derivate din aceeași clasă de bază să răspundă la același apel de metodă în felul lor unic.

Polimorfismul este capacitatea a cel puțin două clase derivate din aceeași clasă de bază să răspundă la același apel de metodă în felul lor unic.

Acest lucru simplifică codul clientului (cel care folosește arhitectura de clase creată de noi), deoarece acesta nu trebuie să se îngrijească de tipul pe care îl referențiază cele două obiecte corespunzătoare claselor derivate atâta timp cât ambele răspund la aceeași comandă.





Să ne imaginăm că stăpânul a două animale de casă vrea să le dresze pe acestea. Să luăm ca exemplu o pisică și un câine.

Stăpânul dorește ca la o comandă câinele să latre și pisica să miaune. Acesta are două variante:

1. Fără polimorfism: Dresează câinele să latre la comanda „LATRĂ!” și pisica să miaune la comanda „MIAUNĂ!”.
2. Cu polimorfism: Dresează ambele animale astfel încât la comanda „EXECUTĂ!” câinele să latre și pisica să miaune.


itacad  
you@technology

## Polimorfism – Implementarea exemplului

```
class Superman
{
    static void Main(string[] args)
    {
        // construim o listă cu animalele pe care le deține Superman
        List<Animal> myPets = new List<Animal>();
        // construim obiectele ce definesc cele două animale ale lui Superman
        Căine rex = new Căine();
        Pisică kitty = new Pisică();
        // le adăugăm în listă
        myPets.Add(rex);
        myPets.Add(kitty);
        // parcurgem lista și executăm comanda
        foreach(Animal pet in myPets)
        {
            // se vor afișa, pe rând, mesajele "HAM" și respectiv "MIAU"
            pet.execută();
        }
    }
}
```

Pentru a realiza ce ne-am propus în slide-ul trecut vom apela la 4 clase:

1. **Animal**
2. **Căine**
3. **Pisică**
4. **Superman** (clasa în care se găsește metoda main și din care se pornește comanda)

```
class Animal
{
    public virtual void execută()
    {
        Console.WriteLine("Fluieră");
    }
}
```

```
class Căine : Animal
{
    public override void execută()
    {
        Console.WriteLine("HAM");
    }
}
```

```
class Pisică : Animal
{
    public override void execută()
    {
        Console.WriteLine("MIAU");
    }
}
```

## Polimorfism – Implementarea exemplului

```
class Superman
{
    static void Main(string[] args)
    {
        // construim o listă cu animalele pe care le deține Superman
        List<Animal> myPets = new List<Animal>();
        // construim obiectele ce definesc cele două animale ale lui Superman
        Câine rex = new Câine();
        Pisică kitty = new Pisică();
        // le adăugăm în listă
        myPets.Add(rex);
        myPets.Add(kitty);
        // parcurgem lista și executăm comanda
        foreach(Animal pet in myPets)
        {
            // se vor afișa, pe rând, mesajele "HAM" și respectiv "MIAU"
            pet.execută();
        }
    }
}
```

```
class Superman
{
    static void Main(string[] args)
    {
        // construim o listă cu animalele pe care le deține Superman
        // lista este o colecție predefinită și se comportă ca un vector de obiecte Animal (în
        acest exemplu)
        List<Animal> myPets = new List<Animal>();


        // construim obiectele ce definesc cele două animale ale lui Superman
        Câine rex = new Câine();
        Pisică kitty = new Pisică();

        // le adăugăm în listă
        myPets.Add(rex);
        myPets.Add(kitty);

        // parcurgem lista și executăm comanda
        foreach(Animal pet in myPets)
        {
            // se vor afișa, pe rând, mesajele "HAM" și respectiv "MIAU"
            pet.execută();
        }
    }
}
```

Acest lucru se întâmplă deoarece:


- chiar dacă „rex” și „kitty” sunt instanțe ale claselor Câine și respectiv Pisică, ele derivă din Animal, prin urmare pot fi adăugate în lista de animale, fără vreo eroare de compilare sau rulare
- la parcurgerea listei cu animale și la executarea metodei, va conta tipul instanței animalului (Câine și respectiv Pisică), de aceea nu se afișează de două ori „Fluieră”, ci „HAM” și „MIAU”.


**itacad**  
 you@technology

## Exemplu de implementare a metodelor

```

class A
{
    public virtual void M() { Console.WriteLine("A"); }
}
class B: A
{
    public override void M() { Console.WriteLine("B"); }
}
class C: B
{
    new public virtual void M() { Console.WriteLine("C"); }
}
class D: C
{
    public override void M() { Console.WriteLine("D"); }
    static void Main()
    {
        D d = new D(); C c = d; B b = c; A a = b;
        d.M(); c.M(); b.M(); a.M();
    }
}
                    
```



36

**Soluția:** Se va afișa DDBB la consolă

### Logica programului

În program, se crează un singur obiect. Acesta este un obiect de tipul **D** creat cu următoarea declarație:

```
D d = new D();
```

Restul declarațiilor din Main declară variabile de tipuri diferite, toate având însă referință către același unic obiect:


- c este de tipul C și primește referință către d
- b este de tipul B și primește referință la c, deci indirect la d
- a este de tipul A și primește referință la b, deci indirect la d

Urmează apoi apelarea metodei M din toate variabilele. În continuare se explică fiecare dintre acestea individual.

Prima instrucțiune:

**d.M();**

Acesta este un apel la **D.M**, metodă ce a fost declarată ca **override** deci implicit virtuală. Acest lucru înseamnă că la run-time este apelată cea mai derivată implementare a lui **D.M** în obiectul de tip **D**. Această implementare este **D.M** care afișază la consolă **D**.

 itacad  
you@technology  
  
Microsoft  
.NET

## Exemplu de implementare a metodelor

```

class A
{
    public virtual void M() { Console.WriteLine("A"); }
}
class B: A
{
    public override void M() { Console.WriteLine("B"); }
}
class C: B
{
    new public virtual void M() { Console.WriteLine("C"); }
}
class D: C
{
    public override void M() { Console.WriteLine("D"); }
    static void Main()
    {
        D d = new D(); C c = d; B b = c; A a = b;
        d.M(); c.M(); b.M(); a.M();
    }
}
    
```

D  
D  
B  
B

37

### c.M();

Acesta este un apel pentru **C.M**, metodă definită ca virtuală. La run-time, se va apela cea mai derivată implementare a unui obiect de tipul **D**. Din moment ce **D.M** suprascrie **C.M**, **D.M** este cea mai derivată implementare în acest caz. De aceea, se va afișa iarăși **D** la consolă.

### b.M();

Acesta este un apel către **B.M** care este declarată override deci implicit virtuală. De aceea, la run-time se va apela cea mai derivată implementare a lui **B.M** într-un obiect de tip **D**. Din moment ce **C.M** nu suprascrie **B.M** ci introduce o metodă nouă ce ascunde **C.M**, cea mai derivată implementare a lui **B.M** într-un obiect de tipul **D** este **B.M**. De aceea, se va afișa **B** la consolă.

### a.M();

Acesta este un apel pentru **A.M**, metodă definită ca virtuală. La run-time, se va apela cea mai derivată implementare a lui **A.M** într-un obiect de tipul **D**. Din moment ce **B.M** suprascrie **A.M**, dar, ca mai sus, **C.M** nu suprascrie **B.M** ci introduce o metodă nouă ce ascunde **C.M**, cea mai derivată implementare a lui **A.M** într-un obiect de tipul **D** este **B.M**. De aceea, se va afișa **B** la consolă.



## Overview

- ▶ Derivarea claselor
- ▶ Folosirea claselor abstracte
- ▶ Folosirea interfețelor
- ▶ Folosirea polimorfismului

