



Modulul 6

Fundamentele POO

➤The more perfect a machine becomes, the more they are invisible behind their function. It seems that perfection is achieved not when there is nothing more to add, but when there is nothing more to take away. At the climax of its evolution, the machine conceals itself entirely.

Antoine de Saint-Exupéry, *Wind, Sand and Stars*

➤The minimum could be defined as the perfection that an artifact achieves when it is no longer possible to improve it by subtraction. This is the quality that an object has when every component, every detail, and every junction has been reduced or condensed to the essentials. It is the result of the omission of the inessentials.

John Pawson, *Minimum*

➤The main aim of communication is clarity and simplicity. Simplicity means focused effort.

Edward de Bono, *Simplicity*



itacad
you@technology

Microsoft®
.NET

Overview

- ▶ Clase și obiecte
- ▶ Utilizarea încapsulării
- ▶ C# și orientarea pe obiecte
- ▶ Definirea sistemelor orientate pe obiecte



2

- C# este un limbaj de programare orientat pe obiecte. În această lecție veți învăța terminologia și conceptele legate de clase în C#.
- După completarea acestui modul studentul va fi capabil să:
 - Definească ce este o clasă și un obiect în contextul programării orientate pe obiecte.
 - Definească aspectele de bază ale unui obiect: identitate, structură, comportament.
 - Descrie abstractizarea și cum aceasta este folosită pentru a crea clase reutilizabile care sunt ușor de menținut.
 - Utilizeze încapsularea pentru a combina metode și date într-o singură clasă.
 - Explice conceptele de moștenire și polimorfism.
 - Creeze și utilizeze clase în C# .

Ce este o clasă?

- ▶ Pentru filosof...
 - ▶ Un rezultat al clasificării umane
 - ▶ Clasificare bazată pe elemente sau atribute comune
 - ▶ Acord în descrierea și denumirea claselor folosite
 - ▶ Creare vocabular; comunicăm!; gândim!
- ▶ Pentru programator...
 - ▶ O construcție sintactică pentru atribute și comportamente comune
 - ▶ O structură de date ce include date și funcții




3

➤ Cuvântul de bază pentru clasificare este **clasa**. Formarea de clase este un act al clasificării și este ceva ce nu numai programatorii fac ci toți oamenii în viața de zi cu zi. De exemplu, toate mașinile au un comportament comun și atribute comune (au patru roți, un motor etc.). Prin urmare, cuvântul **mașină** definește toate aceste comportamente și proprietăți comune ale acestui lucru. Imaginați-vă cum ar fi dacă atributele comune și comportamentele nu ar fi clasificate sub un concept cu o anumită denumire! În loc să spuneți **mașină**, atunci când vă referiți la mașină trebuie să descrieți de fiecare dată proprietățile mașinii pentru a vă putea face înțeleși. Propozițiile ar fi lungi și comunicarea ar fi aproape imposibilă. Atât timp cât fiecare admite că un cuvânt definește ceva anume, și cei care comunică folosesc aceeași limbă, se pot exprima idei complexe și precise într-o formă compactă. Apoi se pot folosi aceste concepte denumite pentru a forma concepte mai complexe și pentru a crește expresivitatea comunicării.

➤ Toate limbajele de programare pot descrie date și funcții de bază. Această abilitate de a descrie proprietăți ne ajută să evităm duplicatele. Codul care se repetă este foarte greu de menținut și crează probleme. Programarea orientată pe obiecte pune acest concept la un nivel superior permițând descrierea de clase (mulțimi de obiecte) care au comportamente și proprietăți comune. Dacă este făcută corespunzător această paradigmă merge foarte bine și se potrivește în întregime cu felul în care oamenii gândesc și comunică.

➤ Clasele nu sunt folosite doar pentru a clasifica obiecte concrete (precum mașinile), ci și obiecte abstracte. Totuși, când sunt clasificate obiecte abstracte, granițele devin mai puțin clare și un *design* corespunzător este și mai important.




➤ Singura idee importantă cu adevărat despre o clasă este că ajută oamenii să comunice.

 itacad
you@technology

Microsoft
.NET

Ce este un obiect?

- ▶ Un obiect este o instanță a unei clase
- ▶ Prezentarea obiectului
 - ▶ Identitate
 - ▶ Fiecare obiect e diferit de altul
 - ▶ Comportament
 - ▶ Obiectele efectuează operații
 - ▶ Structură
 - ▶ Obiectele păstrează informații

4

➤ Cuvântul **mașină** poate avea diferite semnificații în funcție de context. Uneori folosim **mașină** pentru a ne referi la conceptul general, la clasa **mașina**, nu pentru a ne referi la o mașină specifică. Alteori, în schimb, ne referim exact la o anumită mașină, ceva specific. Programatorii folosesc termenul **obiect** sau **instanță** când se referă la o componentă specifică a unei clase. Cele trei caracteristici ale unui obiect sunt tratate în cele ce urmează:

➤ Identitate

- Identitatea este caracteristica prin care un obiect este diferențiat de oricare alt obiect din aceeași clasă. De exemplu, imaginați-vă că doi vecini dețin fiecare câte o mașină la fel (fabricant, model, culoare). Cu toate aceste asemănări, numărul de înmatriculare este singurul care este garantat a fi unic și deosebește cele două mașini. Imaginați-vă cum ar funcționa asigurarea dacă cele două mașini nu s-ar deosebi prin ceva unic.

➤ Comportament

- Comportamentul este caracteristica unui obiect de a fi util. Obiectele există în ideea de a avea un anumit comportament. De cele mai multe ori ignorăm cum mașinile funcționează și ne gândim doar la ceea ce ne oferă nouă direct: ne ajută să ne deplasăm. Funcționarea există dar este inaccesibilă majorității șoferilor, ceea ce este într-adevăr important este comportamentul mașinii, acesta contribuind și la clasificare. O mașină face parte din clasa **mașină** pentru că poți să o conduci, un pix face parte din clasa **pix** pentru că poți scrie cu el.

Comparație clasă/structură

- ▶ Structura este o schiță pentru o valoare
 - ▶ Fără identitate, structură accesibilă, niciun comportament
- ▶ Clasa este o schiță pentru un obiect
 - ▶ Identitate, structură inaccesibilă, comportament

```

struct Time
{
    public int hour;
    public int minute;
}

class BankAccount
{
    ...
    ...
}
    
```

5

➤ Structuri

- O structură precum **Time** în exemplul de mai sus nu are nicio identitate. Dacă avem două variabile de tip **Time** ambele marcând ora 12:30, acestea se vor comporta la fel indiferent pe care o folosim în program. Entitățile software fără identitate se numesc **valori**. Tipurile predefinite descrise în modulul 3 precum **int**, **bool**, **char** și toate tipurile bazate pe **struct** se numesc **tipuri valoare**.
- Variabilele de tipul **struct** pot conține metode dar nu este recomandat. Ele ar trebui să conțină numai date. Cu toate acestea este perfect rezonabil să definești **operatori** pentru o structură. Operatorii sunt metode stilizate care nu adaugă noi funcționalități structurii și oferă o sintaxă mai precisă despre comportamentul acesteia.

➤ Clase

- O clasă precum **BankAccount** în exemplul de mai sus are identitate. Dacă avem două obiecte **BankAccount** programul se va comporta diferit în funcție pe care dintre cele două îl folosim. Entitățile software care au identitate se numesc **obiecte**. Tipurile reprezentate de clase se numesc **tipuri referință** în C#. În contrast cu structurile, la o clasă construită corect trebuie să fie vizibile doar metodele.

➤ Tipuri valoare și tipuri referință

- **Tipurile valoare** sunt cele găsite la nivelul cel mai de jos al unui program. Ele sunt folosite pentru a construi entități software mai complexe. Instanțele de tip valoare pot fi copiate și există pe **stivă** ca variabile locale ori ca attribute în interiorul obiectului pe care îl descriu.
- **Tipurile referință** sunt cele găsite la nivelul înalt al unui program. Ele sunt construite din entități mai mici. Instanțele nu pot fi copiate în general, și sunt păstrate pe **heap**.


itacad
you@technology

Microsoft
.NET

Abstractizarea

- ▶ Abstractizarea este ignorarea selectivă
 - ▶ Decizi ce e important sau nu
 - ▶ Te concentrezi și te bazezi pe ce e important
 - ▶ Ignori și nu te bazezi pe ce e neimportant
 - ▶ Utilizezi încapsularea pentru a impune abstractizarea

The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise. Edsger Dijkstra

6

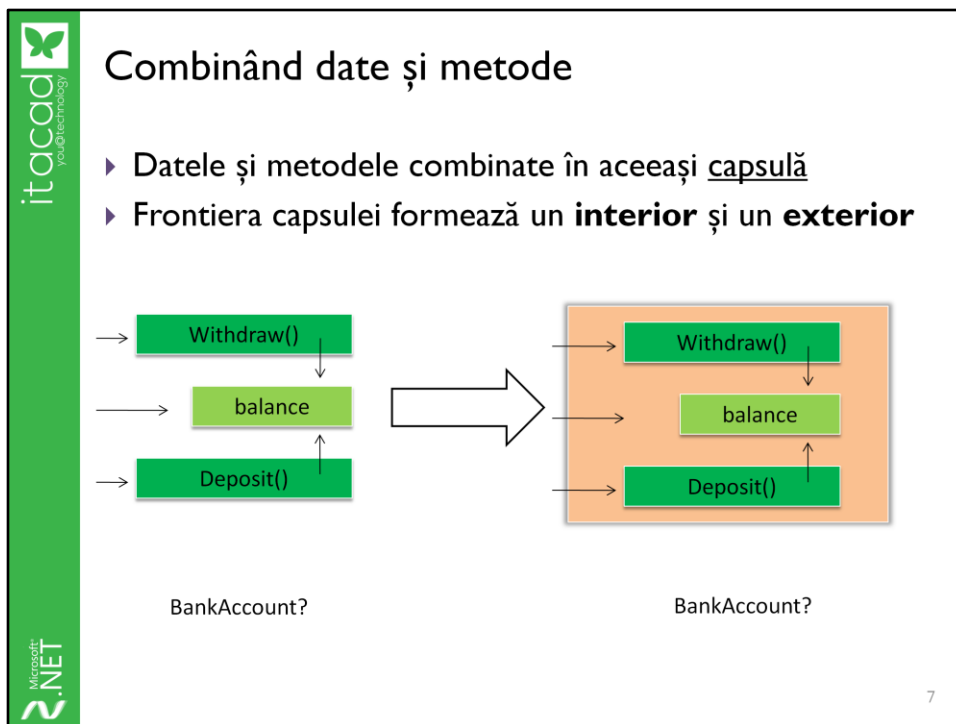
➤ **Abstractizarea** este tactica de a curăța o idee sau un obiect de toate acompaniamentele inutile până când rămâne esențialul, forma minimală. O abstractizare bună dă la o parte toate detaliile neimportante și ne ajută să ne concentrăm doar pe informațiile importante.

➤ **Sa luăm ca exemplu clasa "Mașină". Pentru un pilot de curse sunt importante alte proprietati ale mașinii față de un șofer obișnuit. Șoferul obișnuit ar vrea să știe ce culoare are mașina, în câte secunde atinge suta de metri sau câți decibeli are sistemul audio. În schimb, pentru pilotul de curse sunt importante proprietăți ca: numărul de rotații al motorului pe secundă, gradul de manevrabilitate, gradul de aerodinamicitate, etc.** Astfel, un programator poate proiecta o clasă cu informațiile potrivite pentru utilizatorii acelei clase.

➤ Abstractizarea este un principiu software foarte important. O clasă cu un design bun pune la dispoziție un set minimal de metode care descriu comportamentul esențial al clasei într-o manieră ușor de folosit. Din păcate, abstractizarea software nu este ușoară implicând o înțelegere foarte bună a problemei și a contextului său, claritatea ideilor și foarte multă experiență.

➤ Cele mai bune abstractizări software fac lucrurile mult mai ușoare prin ascunderea lucrurilor care nu sunt esențiale pentru utilizator. Prin urmare elementele neesențiale ale unei clase nu pot fi folosite de către utilizator păstrând o dependență minimă prin metode.

➤ Întotdeauna în dezvoltarea software apare situația în care codul trebuie modificat. Ceea ce trebuie urmărit este ca impactul modificărilor asupra utilizatorului să fie cât mai mic. Cu cât mai puțin se depinde de anumite elemente cu atât impactul modificărilor este mai mic.

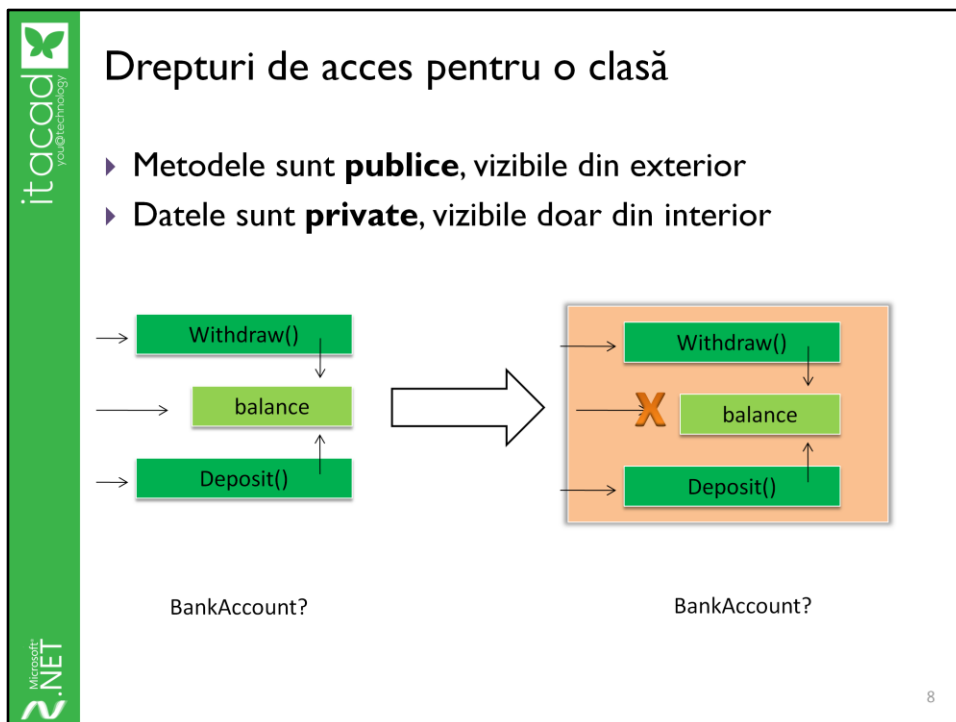


► Programare procedurală

- Programele scrise în limbaje procedurale precum **C** conțin în esență foarte multe date și funcții. Orice funcție poate accesa orice dată. Pentru un program mic lucrurile merg bine, însă când e vorba de o aplicație mare devine mai puțin realizabil. De altfel pe măsură ce programul devine și mai mare modificările sunt aproape imposibile cauzând un impact major.
- Prin păstrarea datelor separate de funcții apare o altă problemă: această tehnică nu corespunde felului în care oamenii gândesc neputându-se realiza abstractizarea.

► Programare orientată pe obiecte

- Toate problemele enumerate mai sus pot dispărea o dată cu folosirea programării orientate pe obiecte.
- Cel mai important pas de la programarea procedurală spre programarea orientată pe obiecte este combinarea metodelor și datelor într-o singură entitate.



➤ În graficul din stânga datele și metodele sunt grupate într-o singură capsulă pe care am numit-o **BankAccount**. Cu toate acestea apare o eroare în acest model: data **balance** este accesibilă din exterior. Imaginați-vă dacă aceste conturi ar fi adevărate iar soldul ar fi direct accesibil clientului, acesta ar putea mări suma din sold fără a realiza niciun depozit. Nu aceasta este modalitatea în care sistemul bancar funcționează și acest model nu prea corespunde.

➤ Această problemă poate fi rezolvată utilizând **încapsularea**. Odată ce datele și metodele sunt combinate într-o singură capsulă, capsula în sine va forma o graniță creând un interior și un exterior. Putem folosi această graniță pentru a controla accesibilitatea entităților din interiorul capsulei: unele sunt accesibile doar din interior – **private**, altele sunt accesibile și din interior și din exterior – **publice**.

➤ Pentru a apropia modelul **BankAccount** de unul adevărat putem să facem metodele **Withdraw()** și **Deposit()** publice, iar **balance** privată. Acum singura metodă de a crește suma din cont este prin depozit. Observați că metoda **Deposit()** poate accesa **balance** pentru că este tot în interior.

➤ C#, ca orice alt limbaj orientat pe obiecte vă oferă flexibilitate în a decide ce membrii ai unei clase să fie accesibili. Puteți, dacă doriți, să creați date publice deși acest lucru nu este recomandat. Datele ar trebui să fie private.

➤ Tipurile ale căror date sunt complet private se numesc **tipuri de date abstracte** (ADTs). Sunt abstracte în sensul în care nu puteți accesa (sau baza pe) datele private ci doar pe metode.

➤ Tipurile predefinite, precum **int**, sunt în felul lor ADTs. Când vreți să adunați două variabile întregi, nu trebuie să știți reprezentarea internă binară a fiecărei variabile, trebuie doar să știți operatorul pentru adunare (+).

itacad
you@technology

Microsoft
.NET

De ce încapsulare?

- ▶ Permite controlul
 - ▶ Obiectul e folosit doar prin metode publice
- ▶ Permite schimbările
 - ▶ Utilizarea obiectului nu se schimbă chiar dacă datele private sunt modificate

➤ Permite controlul

▪ Primul motiv pentru încapsulare este controlul asupra utilizării. Când conduci o mașină te gândești doar la actul condusului nu la componentele interne ale unei mașinii. Când extragi bani din cont nu te gândești la cum este reprezentat contul. Încapsularea și metodele sunt folosite pentru a construi obiecte software care să fie utilizate doar în modul în care dorești.

➤ Permite schimbările

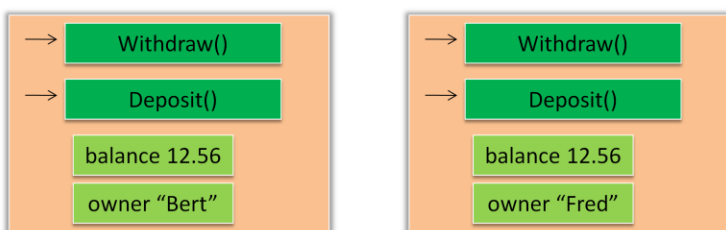
▪ Al doilea motiv pentru utilizarea încapsulării derivă din primul. Dacă implementarea internă a unui obiect este privată, poate fi modificată iar modificările nu afectează modul în care obiectul este folosit (doar prin metodele publice).

▪ Abilitatea de realiza modificări interioare este strâns legată de abstractizare. Fiind date două construcții pentru o metodă, una publică și una privată, întotdeauna alegeți-o pe cea privată.

▪ Cu alte cuvinte, dacă trebuie să alegeți cum să faceți o metodă (publică sau privată) întotdeauna decizia ar trebui să fie privată. O metodă privată poate fi ușor modificată și probabil mai târziu promovată într-o metodă publică. Dar invers, din publică în privată, ar însemna încălcarea codului clientului.

Object Data

- ▶ Object Data descrie informația conținută în obiecte individuale
 - ▶ De exemplu, fiecare cont are **propriul** sold
 - ▶ Dacă două conturi au același sold e doar coincidență

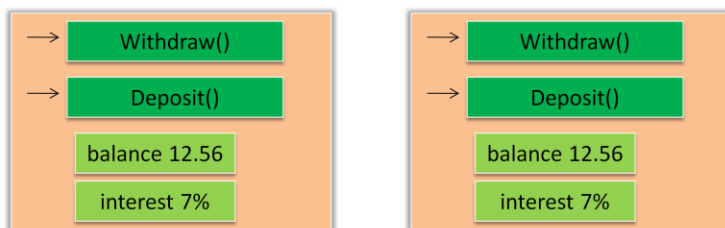


10

- Majoritatea datelor din interiorul unui obiect descriu informații despre acel obiect individual. De exemplu, fiecare cont are propriul sold. Bineînțeles, poate exista și situația în care mai multe conturi au același sold dar asta e doar coincidență: obiectele nu sunt identice!
- Datele din interiorul unui obiect sunt ținute private, și sunt accesibile doar metodelor din interiorul obiectului.

Date statice

- ▶ Datele statice descriu informații despre toate obiectele clasei
 - ▶ Să presupunem că fiecare cont are o dobândă egală
 - ▶ Păstrarea dobânzii în fiecare obiect nu este o idee bună
 - ▶ De ce?



11

➤ Uneori nu are sens să păstrezi informații în interiorul fiecărui obiect. De exemplu, dacă dobânda pentru toate conturile bancare este întotdeauna aceeași, atunci păstrarea informației despre dobândă în fiecare cont ar fi o idee nepotrivită din următoarele motive:

- Este o implementare modestă a problemei descrise prin: "Fiecare cont are aceeași dobândă".
- Crește dimensiunea fiecărui obiect în mod inutil utilizând extra memorie la rularea programului și extra spațiu pe disc la salvarea acestuia.
- E greu de modificat pentru că necesită modificarea fiecărui obiect în parte ceea ce ar putea duce la blocarea tuturor conturilor cât timp au loc schimbările.
- Crește dimensiunea clasei. O dobândă privată ar necesita metode publice prin care să fie accesată

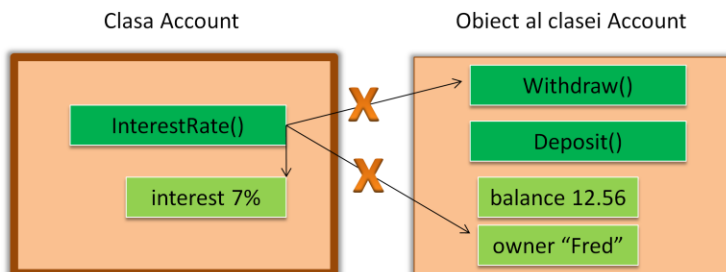
➤ Pentru a rezolva această problemă nu trebuie ca informația comună dintre obiecte să fie repartizată la nivelul fiecărui obiect. Decât să descriem datele comune de fiecare dată la nivelul obiectului, mai bine o facem o singură dată la nivelul clasei. Când informația apare la nivelul clasei automat devine informație globală.

➤ Totuși, prin definiție, datele globale nu sunt păstrate în interiorul clasei și de aceea nu pot fi încapsulate. De aceea, multe limbaje de programare orientate pe obiecte (inclusiv C#) nu permit date globale. În schimb, ele permit ca datele să fie descrise ca statice.

➤ Datele statice sunt declarate în interiorul clasei și beneficiază de încapsularea pe care clasa o pune la dispoziție, dar este asociată cu clasa în sine, nu cu fiecare obiect în parte. Cu alte cuvinte, datele statice sunt declarate în interiorul clasei ca un avantaj sintactic și există chiar dacă programul nu crează niciun obiect al acelei clase.

Metode statice


- ▶ Metodele statice pot accesa doar date statice
 - ▶ O metodă statică este apelată pentru clasă nu pentru obiect



112

- Metodele statice se folosesc pentru a încapsula date statice. În exemplul din slide dobânda aparține clasei, ci nu fiecărui obiect individual. De aceea are sens să declarăm metode la nivelul clasei care să permită accesarea și modificarea dobânzii.
- Metodele se declară statice în același fel în care datele sunt declarate statice. Metodele statice există numai la nivelul clasei. Accesibilitatea la metodele și datele statice se face ca de obicei prin modificatorii **public** și **private**.
- Metoda statică fiind la nivelul clasei este apelată pentru clasă, nu pentru obiect. Ea nu poate accesa date sau metode care sunt nestatice și nu poate fi apelată cu ajutorul cuvântului cheie **this** (va fi explicat într-un slide următor).
- Metodele statice pot accesa entitățile private ale unei clase și pot accesa date nestatice private prin intermediul unui obiect folosit ca referință. Codul următor vă oferă un exemplu:

```
class Time
{
    ...
    public static void Reset(Time t)
    {
        t.hour = 0; // Okay
        t.minute = 0; // Okay
        hour = 0; // compile-time error
        minute = 0; // compile-time error
    }
    private int hour, minute;
}
```

 itacad
you@technology

Microsoft
.NET

Hello, world!

```
using System;
class Hello
{
    public static int Main( )
    {
        Console.WriteLine("Hello, world");
        return 0;
    }
}
```

13

➤ Cum este invocată o clasă la rularea unui program?

- Dacă există o singură metoda **Main()**, aceasta va fi punctul de pornire al programului.

```
// OneEntrance.cs
class OneEntrance
{
    static void Main( )
    {
        ...
    }
}
// end of file
c:\> csc OneEntrance.cs
```

- Totuși dacă există mai multe metode **Main()**, una dintre ele trebuie neapărat să fie construită ca punct de pornire al programului (și acel **Main** trebuie să fie explicit **public**).

```
// TwoEntries.cs
using System;
class EntranceOne
{
    public static void Main( )
    {
        Console.Write("EntranceOne.Main( )");
    }
}
```

Definirea unei clase

- Date și metode împreună într-o clasă
- Metodele sunt publice, datele sunt private

```
class BankAccount
{
    public void Withdraw(decimal amount)
    { ... }
    public void Deposit(decimal amount)
    { ... }
    private decimal balance;
    private string name;
}
```

14

► Deși clasele și structurile sunt semantic diferite, au similarități sintactice. Pentru a defini o clasă față de o structură:

- Utilizați cuvântul cheie **class** în loc de **struct**
- Declarați datele în interiorul clasei la fel ca și la structură
- Declarați metodele în interiorul clasei
- Adăugați modificatorii de accesibilitate pentru date și metode. Cei mai simpli sunt **public** și **private**.

```
class BankAccount
{
    public void Deposit(decimal amount)
    {
        balance += amount;
    }
    private decimal balance;
}
```

► În exemplul de mai sus metoda **Deposit()** poate accesa **balance** pentru că este o metodă a aceleiași clase ca și data **balance**. Cu alte cuvinte **Deposit()** este în interior. Din exterior, membrii privați ai clasei sunt întotdeauna inaccesibili. Exemplu următor va genera o eroare la compilare tocmai din acest motiv:

```
class BankRobber
{
    public void StealFrom(BankAccount underAttack)
    {
        underAttack.balance -= 999999M;
    }
}
```


itacad
you@technology

Microsoft
.NET

Instanțierea unui obiect

- Declarația unei variabile de clasă nu e echivalentă cu instanțierea unui obiect
- Utilizați operatorul **new** pentru a crea un obiect

```
class Program
{
    static void Main( )
    {
        Time now;
        now.hour = 11;

        BankAccount yours = new BankAccount( );
        yours.Deposit(999999M);
    }
}
```

15

- Considerăm următoarele exemple:

```
struct Time
{
    public int hour, minute;
}

class Program
{
    static void Main( )
    {
        Time now;
        now.hour = 11;
        now.minute = 59;
        ...
    }
}
```

- Variabilele de tip **struct Time** sunt variabile de tip valoare. Asta înseamnă că atunci când creezi o astfel de variabilă (cum e **now** în **Main()**) se crează o valoare pe stivă. În acest caz structura **Time** conține doi întregi, deci declarația lui **now** înseamnă crearea a doi întregi pe stivă, unul numit **now.hour**, iar celălalt **now.minute**. Acești întregi nu sunt implicit inițializați cu 0. Prin urmare valorile **now.hour** și **now.minute** nu pot fi citite până când nu le vor fi atribuite valori. Sfera de vizibilitate a acestor variabile valoare este blocul în care au fost declarate, în exemplul de mai sus **Main()**. Aceasta înseamnă că în momentul în care controlul nu mai este acordat metodei **Main** (fie din cauza unei expresii **return** sau aruncarea unei excepții) , **now** va înceta să mai existe.

Cuvântul cheie **this**

- **this** se referă la obiectul care apelează metodele clasei
- Util când identificatorii din diferite domenii sunt în conflict

```
class BankAccount
{
    ...
    public void SetName(string name)
    {
        this.name = name;
    }
    private string name;
}
```

- Dacă în loc de "this.name=name" ar fi "name=name" ce s-ar întâmpla?

16

► Cuvântul cheie **this** se referă implicit la obiectul care apelează metoda ce aparține aceluiași obiect.

► În codul următor instrucțiunea **name = name** nu ar avea niciun efect. Aceasta se întâmplă pentru că identificatorul din partea stângă a atribuirii nu se referă la **name** ca membru al clasei **BankAccount**. Amândoi identificatorii sunt rezolvați de către compilator ca fiind parametrul **name** al metodei **SetName()**. Atenție, compilatorul nu va anunța nicio eroare pentru acest bug!

```
class BankAccount
{
    public void SetName(string name)
    {
        name = name;
    }
    private string name;
}
```

► Această problemă se poate rezolva utilizând **this**, așa cum este ilustrat pe slide. Cuvântul cheie **this** se referă la obiectul curent pentru care metoda este apelată.

► Problema de mai sus mai poate fi rezolvată și modificând numele parametrului ca în exemplul următor:

```
class BankAccount
{
    public void SetName(string newName)
    {
        name = newName;
    }
    private string name;
}
```




Creare clase imbricate

- ▶ Clasele pot fi declarate în interiorul altor clase
 - ▶ Numele complet al clasei din interiorul altei clase include și numele clasei exterioare

```
class Program
{
    static void Main( )
    {
        Bank.Account yours = new Bank.Account( );
    }
}
class Bank
{
    ... class Account { ... }
}
```

17

- Există cinci tipuri diferite în C#: **class**, **struct**, **interface**, **enum** și **delegate**. Puteți să adunați toate aceste tipuri în interiorul unei clase sau al unei structuri.
- În codul următor, clasa **Account** este imbricată în clasa **Bank**. Numele întreg al acestei clase este **Account.Bank** și acest nume trebuie folosit în exteriorul clasei **Bank**.

```
// Program.cs
class Program
{
    static void Main( )
    {
        Account yours = new Account( ); // compile-time error
    }
}
// end of file
c:\> csc Program.cs
error CS0246: The type...'Account' could not be found...
```

- În contrast doar **Account** poate fi folosit în interiorul clasei **Bank** ca în exemplul următor:

```
class Bank
{
    class Account { ... }
    Account OpenAccount( )
    {
        return new Account( );
    }
}
```

Utilizare clase imbricate

- Clasele interioare pot fi publice sau private

```
class Bank
{
    public class Account { ... }
    private class AccountNumberGenerator { ... }
}
class Program
{
    static void Main( )
    {
        Bank.Account accessible;
        Bank.AccountNumberGenerator inaccessible;
    }
}
```

18

- Puteți controla accesibilitatea unei metode sau a unor date declarându-le publice sau private. Exact în același fel se poate controla și accesibilitatea claselor imbricate.
- O clasă imbricată publică nu are nicio restricție de accesare.
- O clasă imbricată privată se comportă la fel ca o metodă sau o dată privată: este inaccesibilă din exterior. Exemplu:

```
class Bank
{
    private class AccountNumberGenerator
    {
        ...
    }
}
class Program
{
    static void Main( )
    {
        // Compile time error
        Bank.AccountNumberGenerator variable;
    }
}
```

- În acest exemplu **Main()** nu poate folosi **Bank.AccountNumberGenerator** pentru că este o metodă a clasei **Program** și **AccountNumberGenerator** este privată, prin urmare putând fi folosită doar din interiorul clasei **Bank**.



itacad
you@technology

Microsoft
.NET

Moștenirea

- ▶ Moștenirea specifică o relație între clase
- ▶ Noi clase detaliază clasele existente

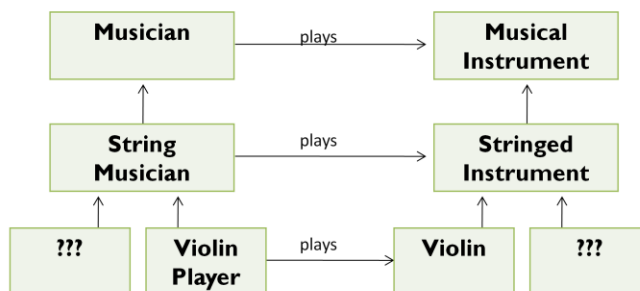


19

- Moștenirea este o relație care este specificată la nivelul clasei. O nouă clasă poate fi derivată dintr-o clasă deja existentă. În exemplul din slide clasa **Violin Player** este derivată din clasa **Musician**. Clasa **Musician** este numită clasă de bază (sau mai rar clasă părinte sau superclasă); clasa **Violin Player** este numită clasă derivată (sau mai rar clasă copil sau subclasă).
- Moștenirea este o relație foarte importantă pentru că o clasă derivată moștenește totul de la clasa de bază. De exemplu, dacă **Musician** conține o metodă **TuneYourInstrument()** atunci această metodă va exista ca membru și în clasa **Violin Player**.
- O clasă de bază poate avea oricâte clase derivate. De exemplu, noi clase precum **FlutePlayer** sau **PianoPlayer** pot fi derivate din **Musician**. Aceste noi clase vor moșteni și ele metoda **TuneYourInstrument()** din clasa de bază.
- Orice schimbare care are loc în clasa de bază automat se va reflecta și în clasele derivate. Prin urmare dacă un bug este introdus în clasa de bază acesta se va regăsi și în clasele derivate.

Ierarhia de clase

- Clasele relaționate prin moștenire formează o ierarhie



20

- Clasele care derivă dintr-o clasă de bază pot la rândul lor să fie derivate obținând noi clase. De exemplu, în slide, **StringMusician** este derivată din **Musician** dar este și o clasă de bază pentru **ViolinPlayer**. Un grup de clase relaționate prin moștenire formează o **ierarhie de clase**. Cum urci în ierarhie, clasele prezintă concepte generale; cum cobori în ierarhie clasele prezintă concepte specifice.
- Adâncimea unei ierarhii de clase este numărul de nivele de moștenire în acea ierarhie. Cu cât mai adâncă este ierarhia cu atât este mai greu de folosit și implementat. Recomandările sunt ca adâncimea să fie cuprinsă între cinci și șapte clase.
- Slide-ul prezintă două ierarhii în paralel: unul pentru **Musician**, celălalt pentru **Musical Instrument**. Crearea de ierarhii nu este ușoară: clasele de bază trebuie stabilite de la început. Moștenirea este o componentă dominantă a *framework*-urilor (modele de lucru care pot fi construite și extinse).

Moștenirea singulară și multiplă

- ▶ Moștenirea singulară
 - ▶ Derivă dintr-o singură clasă de bază
- ▶ Moștenirea multiplă
 - ▶ Derivă din două sau mai multe clase de bază



21

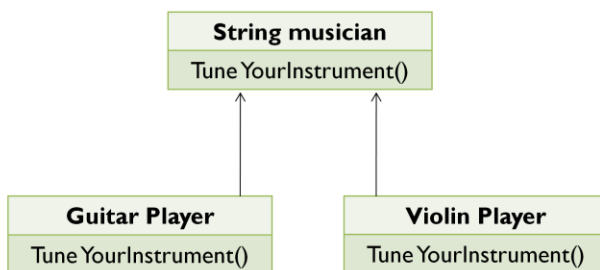
➤ **Moștenirea singulară** este atunci când o clasă derivată are o singură clasă de bază. În exemplul din slide, **Violin** derivă dintr-o singură clasă **StringedInstrument** fiind o moștenire singulară. **StringedInstrument** derivă din două clase de bază, dar nu are nicio relevanță pentru clasa **Violin**.

➤ **Moștenirea multiplă** apare atunci când o clasă derivă din două sau mai multe clase de bază. Exemplul din slide pentru moștenire multiplă este **StringedInstrument** care derivă din **Musical Instrument** și **Pluckable**. Moștenirea multiplă este cea mai susceptibilă la greșeli. C#, ca majoritatea limbajelor de programare moderne (nu și C++) restricționează folosirea moștenirii multiple: poți să moștenești din oricâte interfețe dorești, dar poți moșteni doar dintr-o singură non-interfață (cel mult o clasă abstractă sau concretă). Termenii interfață, clasă abstractă, clasă concretă vor fi acoperiți în slide-urile următoare.

➤ Observați că moștenirea, dar mai ales cea multiplă, oferă mai multe puncte de vedere asupra unui obiect. De exemplu, un obiect **Violin** poate fi folosit la nivelul clasei **Violin**, dar poate fi folosit și la nivelul clasei **StringedInstrument**.

Polimorfism

- ▶ Numele metodei apare în clasa se bază
- ▶ Implementarea metodei apare în clasele derivate
 - ▶ O metodă fără implementare se numește operație



22

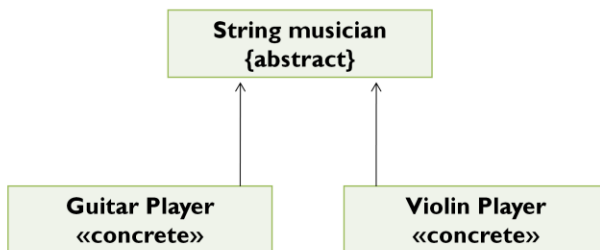
➤ Literal, **polimorfism** înseamnă “mai multe forme”. În POO (Programare Orientată pe Obiecte), definește faptul că o metodă dintr-o clasă de bază poate fi implementată în diferite feluri în clasele derivate.

➤ Considerați scenariul următor: o orchestră care se pregătește pentru un concert și fiecare muzician trebuie să își acordeze instrumentele. Fără polimorfism ar însemna că dirijorul trebuie să viziteze fiecare muzician în parte, să observe la ce instrument cântă și să îi dea indicații despre cum să își acordeze instrumentul. Cu polimorfism, dirijorul va spune doar “Acordați-vă instrumentele” și fiecare muzician va înțelege despre ce este vorba. Dirijorul nu va trebui să știe fiecare instrument la care cântă membrii orchestrei, doar că muzicienii vor răspunde la cerere în maniera corespunzătoare fiecărui instrument. În loc să fie dirijorul cel care are toate cunoștințele, acestea sunt împărțite corespunzător: un chitarist va ști să acordeze chitara, un violonist va acorda vioara. De fapt, dirijorul nu trebuie să știe cum să acordeze niciun instrument. Această alocare descentralizată a responsabilităților înseamnă că noile clase derivate pot fi adăugate ierarhiei fără să fie nevoie să se modifice clasele deja existente.

➤ Apare totuși o problemă: Cum este implementat corpul metodei în clasa de bază? Fără a ști fiecare instrument în parte este imposibil să știi cum să îl acordezi. Pentru a rezolva această problemă doar numele metodei (fără corp) este declarat în clasa de bază. Numele metodei fără corp se numește **operație**(operation).

Clase de bază abstracte

- Unele clase există doar pentru a fi derivate
 - Nu are sens să creezi instanțe ale acestor clase
 - Aceste clase sunt **abstracte**



23

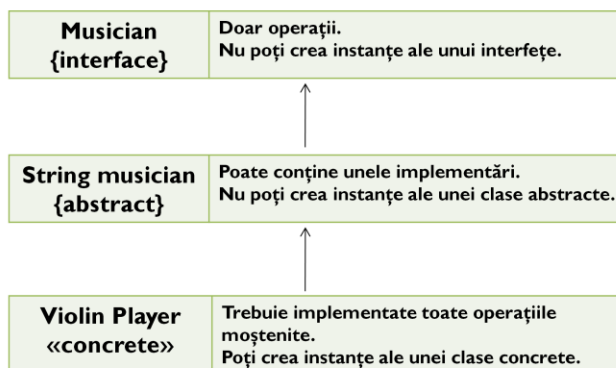
- Într-o ierarhie de clase obișnuită, operația (numele clasei) este declarată în clasa de bază și implementată în diferite forme în clasele derivate. Clasa de bază există doar pentru a introduce numele metodelor în ierarhie. În particular, clasele de bază nu necesită nicio implementare. Prin urmare, este vital ca o clasă de bază să nu fie folosită ca o clasă obișnuită. Mai important, nu ar trebui să îi fie permis utilizatorului să creeze instanțe ale clasei de bază. Mecanismul care permite implementarea restricțiilor sugerate în acest paragraf se numește **abstractizarea clasei**.
- Clasa de bază poate fi marcată ca abstractă utilizând cuvântul cheie **abstract** scris între acolade. În contrast, se poate folosi cuvântul **concrete** pentru a arăta că o clasă poate fi instanțiată.


 itacad
 you@technology

 Microsoft
 .NET

Interfețe

- Interfețele conțin doar operații, nu implementări



24

- Clasele abstracte și interfețele sunt asemănătoare prin faptul că nu pot fi instanțiate. Totuși, diferența majoră dintre ele este că o clasă abstractă poate conține implementări pe când o interfață nu. Se poate spune că o interfață este și mai abstractă decât o clasă abstractă.
- Pentru specificarea unei interfețe se va folosi cuvântul cheie **interface**.
- Interfețele sunt construcții foarte importante în programarea orientată pe obiecte. Ele impun și o anumită notație și terminologie. Când derivezi dintr-o interfață spunem că implementăm interfața respectivă. Când derivezi dintr-o non-interfață (clasă abstractă sau concretă) spunem că extindem acea clasă.
- Întotdeauna interfețele se vor afla în vârful ierarhiei.



Sumar

- ▶ Clase și obiecte
- ▶ Utilizarea încapsulării
- ▶ C# și orientarea pe obiecte
- ▶ Definirea sistemelor orientate pe obiecte

25

➤ Întrebări:

- Explicați conceptul de abstractizare și specificați de ce este important în ingineria software.
- Care sunt cele două principii ale încapsulării?
- Descrieți moștenirea în contextul programării orientate pe obiecte.
- Ce este polimorfismul?
- Descrieți diferențele dintre interfețe, clase abstracte și clase concrete.