



# Programare orientată pe obiecte

---

1. Despre curs
2. Concepte și paradigme în POO



# Despre curs

---

## ■ Cine:

- [Andrei.Vatavu@cs.utcluj.ro](mailto:Andrei.Vatavu@cs.utcluj.ro) – prelegeri
- [Anca.Ciurte@cs.utcluj.ro](mailto:Anca.Ciurte@cs.utcluj.ro) – prelegeri

- J2SE – SDK: Java SDK Oracle
- Eclipse



# Obiectivele cursului

- Să vă ofere cunoștințe operaționale în dezvoltarea software folosind orientarea pe obiecte, cu accent pe conceptele, paradigmele și atitudinile necesare scrierii de cod orientat pe obiecte de calitate
- Să vă permită să vă dezvoltați îndemânarea în programare și inginerie software, să învățați să utilizați efectiv un limbaj de programare orientat pe obiecte relevant din punctul de vedere al industriei software



# Rezultatele cursului (1)

---

- Privitoare la cunoștințe/înțelegere
  - Fundamentele POO
  - Elementele principale în proiectarea, programarea, testarea și documentarea soluțiilor OO
  - Metode de proiectare pentru programe Java de complexitate relativ redusă
  - Elemente de bază ale UML



## Rezultatele cursului (2)

---

- **Abilități intelectuale**
  - Să înțelegeți specificația unei probleme și să o rezolvați printr-un program pentru calculator în paradigma programării orientate pe obiecte.
  - Să luați o specificație parțială și să luați deciziile corespunzătoare asupra funcționalității sistemului propus.
  - Să dezvoltați o specificație în UML dintr-o descriere în limbaj natural



## Rezultatele cursului (3)

---

### ■ Abilități practice

- Să folosiți eficient conceptele de programare OO în Java
- Să dezvoltați programe de complexitate relativ redusă în Java
- Să depanați, testați și documentați soluții folosind orientarea pe obiecte
- Să dezvoltați applet-uri și componente GUI relativ simple



# Subiecte abordate

- Concepte și paradigme în programarea orientată pe obiecte
- Abstracțiuni și tipuri de date abstracte
- Caracteristicile limbajului Java
- Tipurile de date primitive și structurile de control în Java
- Clase și obiecte
- Interfețe Java
- Excepții și tratarea lor
- Principalele API-uri și clase predefinite în Java
- Reprezentarea în UML a claselor, obiectelor și asociațiilor; diagrame de clase și obiecte
- Programarea bazată pe clase și obiecte
- Moștenirea și polimorfismul
- Programarea orientată pe obiecte
- Scurtă introducere în programarea condusă de evenimente
- Scurtă introducere în programarea applet-urilor și a componentelor GUI



# Evaluare. Referințe

## ■ Evaluare

- Evaluarea activității de la laborator (L), evaluarea unui mini-proiect (P).
- Nota finală =  $0.6E + 0.4(L+P)/2$

## ■ Referințe

- **Bruce Eckel, *Thinking in Java*, 4th edition, Prentice Hall, 2006**
- **David J. Barnes & Michael Kölling, *Objects First with Java. A Practical Introduction using BlueJ*, 5th Edition, Prentice Hall / Pearson Education, 2012**
- **Paul & Harvey Deitel, *Java. How to Program*, ed. IX, Prentice Hall, 2012**
- Documentația Java de la Oracle
- Tutoriale Java de la Oracle
- Tutoriale UML introductive
- Documentația BlueJ și Netbeans/Eclipse





# Evaluare. Referințe

- Referinte:
  - [www.udacity.com](http://www.udacity.com) – Introduction into Java Programming
  - <https://docs.oracle.com/javase/tutorial/>
  - <http://www.tutorialspoint.com/java/>



# Concepte și paradigme în POO

- Temele de astăzi
  - Paradigme de programare (programarea imperativă și programarea structurată)
  - Abstractizarea datelor
    - Tipuri de date abstracte
  - Paradigma POO
  - Concepte POO
    - Obiecte și clase
    - Încapsulare și moștenire



# Paradigme de programare

- **Paradigmă** (dict.): Un set de supoziții, concepte, valori și practici care constituie o viziune a realității pentru comunitatea care le adoptă, în special într-o disciplină intelectuală.
  - **Paradigmă de programare**
    - Un model care descrie esența și structura computației
    - Oferă (și determină) viziunea pe care o are programatorul asupra execuției programului
- Exemple:
- În POO, programatorii pot concepe programele ca fiind o colecție de obiecte care interacționează
  - În programarea funcțională un program poate fi conceput ca fiind o secvență de evaluări de funcții, fără stări



# Paradigme de programare

---

Programarea OO

Abstractizarea datelor

Programarea structurată

Programarea imperativă

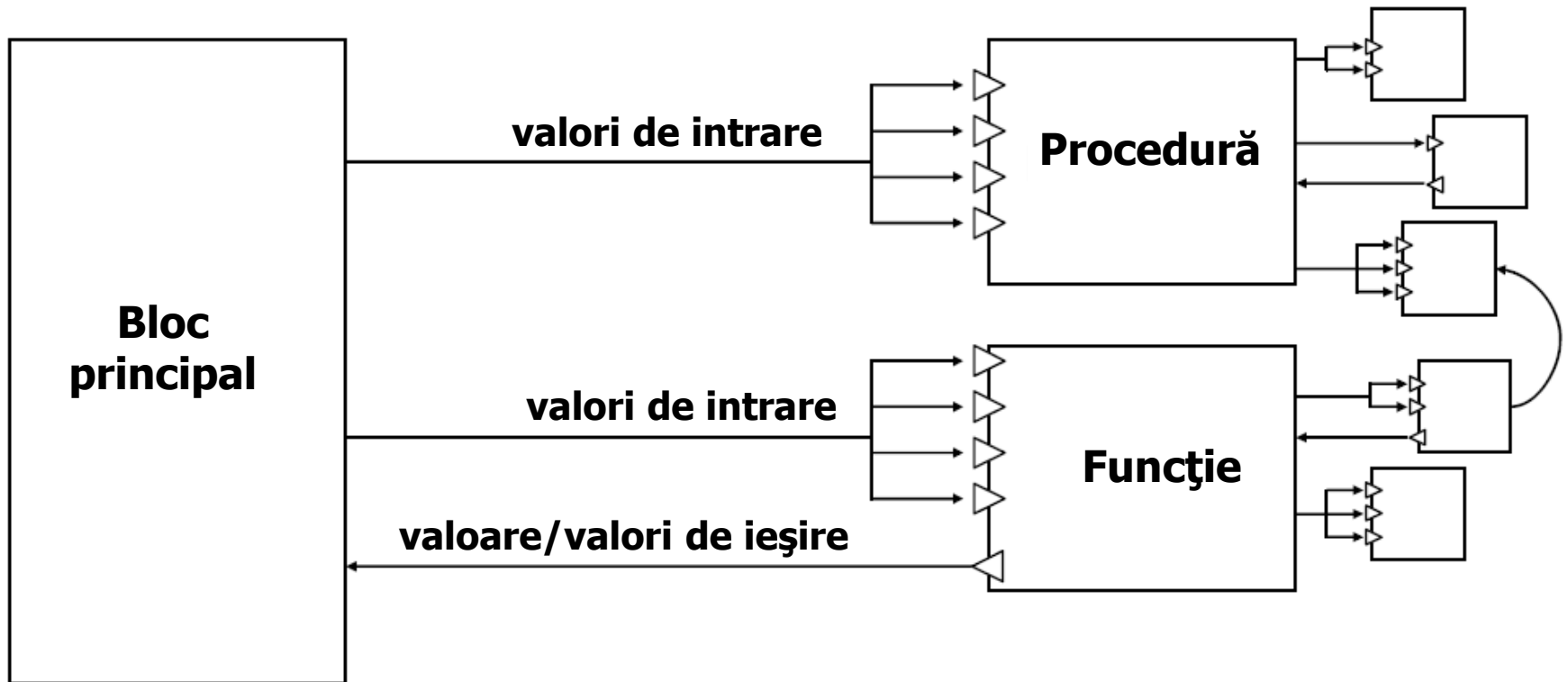


# Programarea imperativă

- În modelul tradițional von Neumann, un calculator constă dintr-o unitate centrală de prelucrare și memorie și el efectuează secvențe de instrucțiuni atomice care accesează, operează asupra valorilor stocate la locații de memorie adresabile individual și le modifică. Aici, o **compuție este o serie de operații aritmetice și de efecte laterale, cum sunt atribuirile sau transferurile de date care modifică starea unității de stocare, intrarea sau ieșirea**
- Ne referim la acest ca ***paradigmă imperativă sau procedurală***.
- Este de subliniat importanța atribuirilor și a variabilelor pe post de containere pentru paradigma imperativă
- Exemple de limbaje: Fortran, Pascal, C, Ada.



# Programarea structurată





# Programarea structurată

- **Abstractizarea operațiilor**
  - **Structura unui modul**
    - **Interfața**
      - Date de intrare
      - Date de ieșire
      - Descrierea funcționalității
    - **Implementarea**
      - Date locale
      - Secvențe de instrucțiuni
  - **Sintaxa limbajului**
    - Organizarea codului în blocuri de instrucțiuni
    - Definiții de funcții și proceduri
    - Extinderea limbajului cu noi operații
    - Apeluri la proceduri și funcții noi



# Beneficiile programării structurate

---

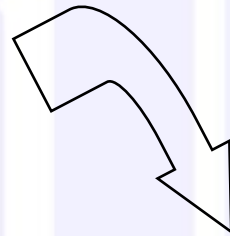
- Ușurează dezvoltarea software
  - Evită repetarea realizării aceluiași lucru
  - Munca de programare este descompusă în module independente
  - Proiectare Top-down: descompunerea în subprobleme
- Facilitează întreținerea software
  - Codul este mai ușor de citit
  - Independența modulelor
- Favorizează reutilizarea software





# Programarea structurată. Exemplu

```
int main()
{
    double u1, u2, m;
    u1 = 4;
    u2 = -2;
    m = sqrt (u1*u1 + u2*u2);
    printf("%lf\n", m);
    return 0;
}
```



```
double module(double u1, double u2)
{
    double m;
    m = sqrt (u1*u1 + u2*u2);
    return m;
}
int main()
{
    printf("%lf\n", module(4, -2));
    return 0;
}
```



# Abstractizarea datelor

- Abstractizarea datelor: impunerea unei separări clare între *proprietățile abstracte* ale unui *tip de dată* și *detaaliile concrete* ale *implementării* lui
  - Proprietăți abstracte: acelea care sunt vizibile codului client care folosește tipul de dată – *interfața* cu tipul de dată
  - Implementarea concretă este păstrată în totalitate privată și ea se poate într-adevăr schimba, spre exemplu pentru a incorpora îmbunătățiri ale performanțelor în timp.



# Tipuri abstracte de date

- Abstractizarea datelor + abstractizarea operațiilor
  - Un tip de dată abstract:
    - **Structură de date** care stochează informații pentru a reprezenta un anumit concept
    - **Funcționalitate:** set de operații care pot fi aplicate tipului de dată
  - Sintaxa limbajului
    - Modulele sunt asociate tipurilor de date
    - Sintaxa nu este neapărat nouă față de programarea modulară



## Exemplu de tip de dată abstract în C

```
struct vector {  
    double x;  
    double y;  
}  
void construct (vector *v, double v1, double v2)  
{  
    v->x = v1;  
    v->y = v2;  
}  
double module(vector v)  
{  
    double m;  
    m = sqrt (v.x*v.x + v.y*v.y);  
    return m;  
}
```

```
int main()  
{  
    vector v;  
    construct(&v, 4, 2);  
    printf("%lf\n", module(v));  
    return 0;  
}
```



# Extensibilitatea tipului de dată abstract

```
...  
  
double product(vector v, vector w) {  
    return v.x*w.x + v.y*w.y;  
}  
  
int main() {  
    vector v;  
    construct(&v, 4, 2);  
    construct(&w, -1, 7);  
    printf("%lf\n", product(v, w));  
    return 0;  
}
```



## Beneficiile tipurilor de date abstracte

---

- Conceptele din domeniu sunt reflectate în cod
- *Încapsulare*: complexitatea internă, datele și detaliile operațiilor sunt ascunse
- Utilizarea tipului de dată este *independentă de implementarea sa internă*
- Oferă o mai mare modularitate
- Sporește ușurința întreținerii și reutilizării codului



# Paradigma orientată pe obiecte

---

- Paradigma programării structurate a avut inițial succes (1975-85)
  - Dar a început să eșueze la produse mai mari (> 50,000 LOC)
- PS avea probleme de întreținere post-livrare (astăzi această întreținere necesită, de la 70 la 80% din efortul total)
- Motivul: Metodele structurate sunt fie
  - orientate pe operații (analiza fluxului de date) fie
  - orientate pe atribute (d.e. dezvoltarea cu metoda Jackson)...
  - ...dar nu amândouă



# Paradigma orientării pe obiecte

---

- Paradigma folosită în limbaje: o simulare a domeniului unei probleme prin abstractizarea informațiilor de comportament și stare din obiecte din lumea reală
- Conceptele de obiecte, clase, transmitere de mesaje și moștenire sunt cunoscute ca făcând parte din paradigma orientării pe obiecte.





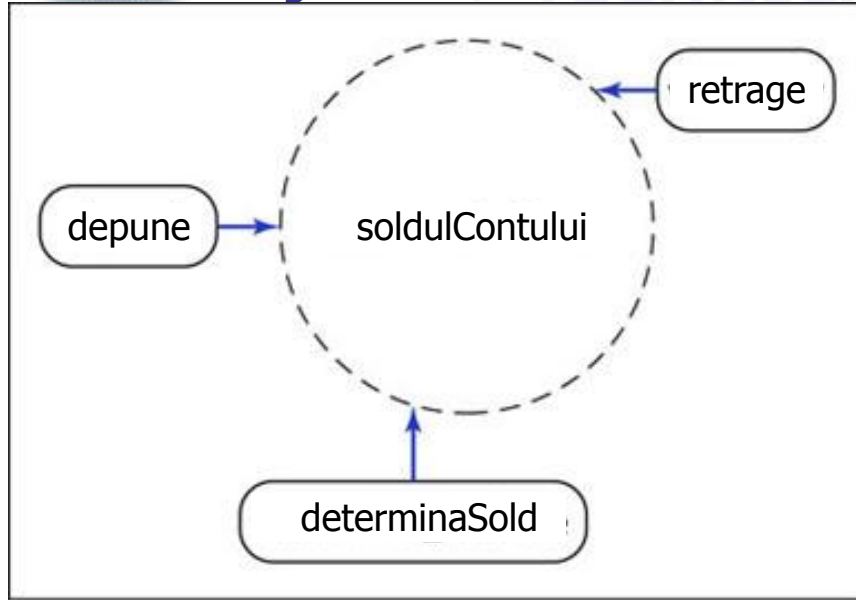
# Paradigma orientării pe obiecte

---

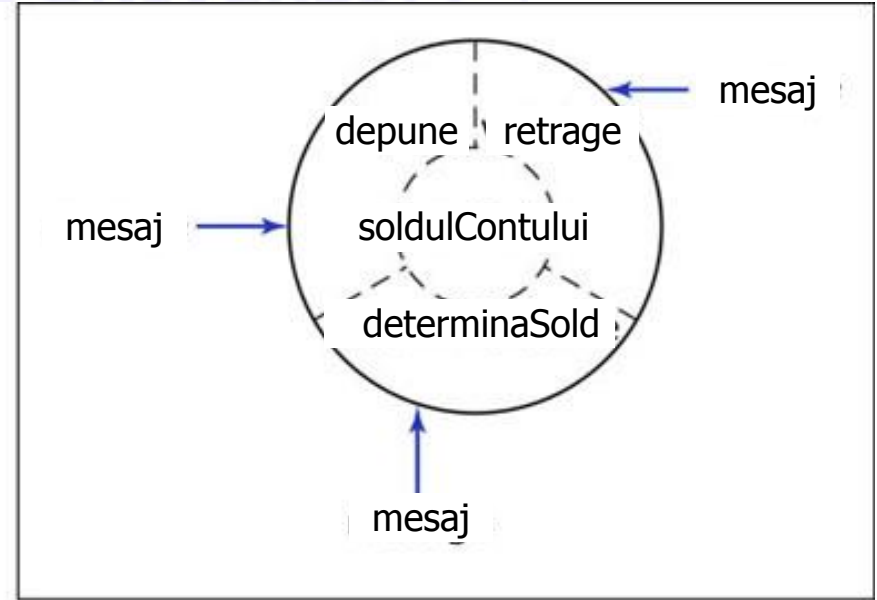
- POO consideră că atât atributele cât și operațiile au importanță egală
- O viziune simplistă a unui obiect poate fi:
  - Obiect = componentă software care încorporează atât atributele cât și operațiile care se pot efectua asupra atributelor și care suportă moștenirea.
- Exemplu:
  - Cont bancar
    - Date: soldul contului
    - Acțiuni: depune, retrage, determină soldul



# Comparație între paradigma structurată și cea obiectuală



(a)



(b)

- Ascunderea informației
- Proiectarea dirijată de responsabilități
- Impact asupra întreținerii și dezvoltării



# Ascunderea informației

---

- În versiunea orientată pe obiecte
  - Linia continuă din jurul lui `soldulContului` arată că în afara obiectului nu se știe cum este implementat `soldulContului`
- În versiunea clasică
  - Toate modulele au detalii privind implementarea lui `soldulContului`



# Punctele tari ale paradigmei OO

---

- Ascunderea informației => Întreținerea post-livrare este mai sigură
  - Șansele apariției erorilor regresive sunt reduse (în software nu se repetă erori cunoscute)
- Dezvoltarea este mai ușoară
  - Obiectele au în general corespondente fizice =>
    - Simplifică modelarea (un aspect cheie al paradigmei OO)



## Punctele tari ale paradigmei OO (2)

---

- Obiectele bine proiectate sunt unități independente
  - Tot ce se referă la obiectul real modelat este în obiect — *încapsulare*
  - Comunicarea se face prin schimb de *mesaje*
  - Această independență este augmentată prin *proiectarea dirijată de responsabilitate*



## Punctele tari ale paradigmei OO (3)

---

- Produs clasic: conceptual o singură unitate (deși poate fi implementată ca un set de module)
  - Paradigma OO reduce complexitatea deoarece produsul constă, în general, din unități independente
- Paradigma OO promovează reutilizarea
  - Obiectele sunt *entități independente*



# Programarea orientată pe obiecte

---

- Oferă
  - suport sintactic pentru tipurile de date abstracte
  - facilități asociate cu ierarhiile de clase
- Schimbă punctul de vedere: programele sunt apendice ale datelor
- Introduce un concept nou: *obiect* = tip de dată abstract cu *stare* (atribute) și *comportament* (operații)



# Concepte OOP

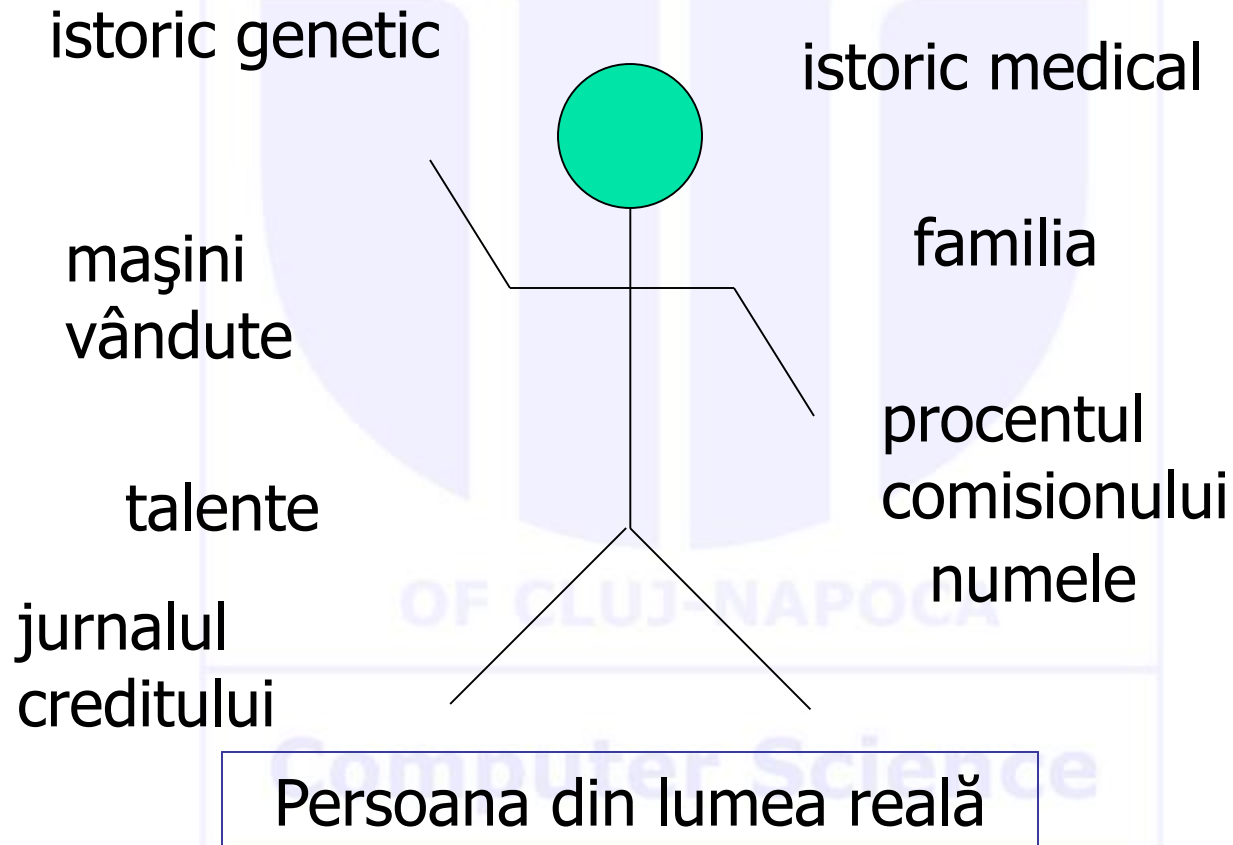
---

- Abstractizare
  - Ne ocupăm doar de datele care prezintă interes pentru problema noastră.
  - *Abstractizare*: proces de filtrare a detaliilor neimportante ale obiectului astfel încât să rămână doar caracteristicile importante
- Exemplu (urmează)



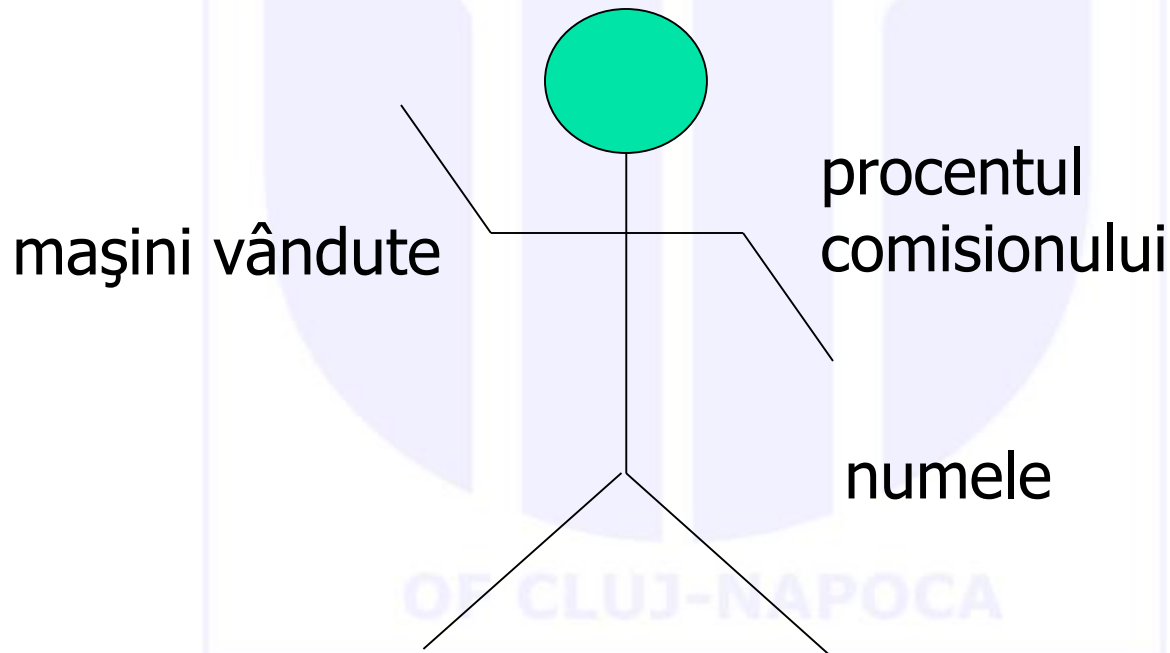


# Abstractizare. Exemplu (1)





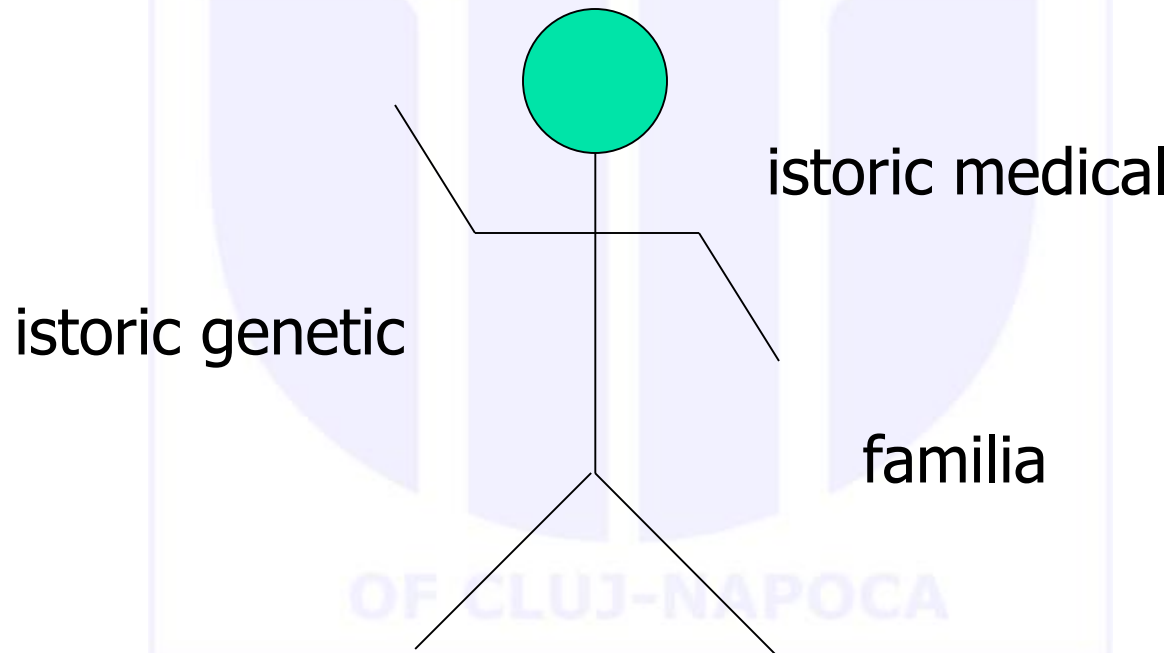
# Abstractizare. Exemplu (2)



Abstractizarea unui tip  
**PersoanaVinzator** pentru un  
Sistem de urmărire a vânzărilor



# Abstractizare



Abstractizarea unui tip  
**Pacient** dintr-o baze de  
date medicală



# Ce sunt obiectele software?

- Blocurile de construcție a sistemelor software
  - Program = colecție de obiecte care interacționează
  - Obiectele cooperează pentru a finaliza o sarcină
    - pentru aceasta, ele comunică trimițându-și "mesaje" unul altuia
- Obiectele modelează lucruri *tangibile*
  - Persoană
  - Bicicletă
  - Cal
  - Bancă etc.



# Ce sunt obiectele software?

- Obiectele modelează *lucruri conceptuale*
  - întâlnire
  - dată calendaristică
- Obiectele modelează *procese*
  - aflarea drumului printr-un labirint
  - sortarea unui pachet de cărți de joc
- Obiectele au
  - *capabilități*: ce pot face, cum se comportă
  - *proprietăți*: trăsături (caracteristici) care descriu obiectele



# Capabilitățile obiectelor: acțiuni

- Obiectele au *capabilități (comportamente)* care le permit să efectueze acțiuni specifice
  - obiectele sunt deștepte — ele "știu" cum să facă anumite lucruri
  - un obiect face ceva *doar dacă un alt obiect îi spune* să-și folosească una dintre capabilități
- Capabilitățile pot fi:
  - *constructori*: stabilesc starea inițială a proprietăților obiectului
  - *comenzi*: modifică proprietățile obiectului
  - *interogări*: furnizează răspunsuri bazate pe proprietățile obiectului



# Capabilitățile obiectelor: acțiuni

---

- Exemple: *borcanele cu gem* sunt capabile să efectueze acțiuni specifice
  - **constructor:** să fie creat
  - **comenzi:** adaugă gem, golește-te
  - **interogări:** răspunde dacă este închis sau deschis capacul, dacă borcanul este plin sau gol



# Proprietățile obiectului: starea

---

- **Proprietățile** : determină cum acționează un obiect
  - Pot fi constante (nu se schimbă) sau variabile
  - Pot fi ele însele obiecte — pot primi mesaje
  - Ex. Capacul *borcanului cu gem* și gemul în sine sunt obiecte
- Proprietățile pot fi:
  - **attribute**: lucruri care ajută la descrierea unui obiect
  - **componente**: lucruri care sunt "parte a" unui obiect
  - **asocieri**: lucruri despre care știe un obiect, dar care nu sunt parte a aceluși obiect





# Proprietățile obiectului: starea

---

- **Stare**: colecție a tuturor proprietăților obiectului; se schimbă dacă o proprietate se schimbă
  - unele nu se schimbă, d.e. volanul unei mașini
  - altele se schimbă, d.e. culoarea mașinii
- Exemplu: proprietățile *borcanelor cu gem*
  - **attribute**: culoare, material, miros
  - **componente**: capac, container, etichetă
  - **asocieri**: un *borcan cu gem* poate fi asociat cu încăperea în care se află



# Clase și instanțe

- Concepția noastră curentă: fiecare obiect corespunde direct unui *anumit* obiect din realitate, d.e., un atom sau un automobil anume
- Dezavantaj: mult prea nepractic să lucrăm cu obiecte în acest fel deoarece
  - ele pot fi infinit de multe
  - nu dorim să descriem fiecare individ separat, deoarece indivizii au multe lucruri în comun
- Clasificarea obiectelor scoate în evidență ce este comun între mulțimi de obiecte similare
  - mai întâi să *descriem ce este comun*
  - apoi să "ștampilăm" oricâte copii



# Clase ale obiectelor

## ■ *Clasa unui obiect*

- categoria obiectului
- definește capabilitățile și proprietățile comune unei mulțimi de obiecte individuale
  - toate *borcanele cu gem* se pot deschide, închide și goli
- definește un șablon pentru crearea de *instanțe de obiect*
  - unele *borcane cu gem* pot fi din plastic, pot fi colorate, de o anumită mărime etc.



# Clase ale obiectelor

- Clasele implementează capabilitățile ca *metode*
  - secvențe de instrucțiuni în Java
  - obiectele cooperează trimițând mesaje altor obiecte
  - fiecare mesaj "invocă o metodă"
- Clasele implementează proprietățile ca *variabile instanță*
  - locație de memorie alocată obiectului, care poate păstra o valoare care se poate schimba



# Instanțe de obiect

- *Instanțele obiectelor* sunt obiecte individuale
  - realizate din șablonul clasei
  - o clasă poate reprezenta un număr nedefinit de instanțe de obiect
  - realizarea unei instanțe de obiect constituie *instanțierea* obiectului respectiv
- Prescurtare:
  - **clasă**: clasa obiectului
  - **instanță**: instanța obiectului (a nu se confunda cu variabilele instanță)



# Instanțe de obiect

- Instanțe diferite ale, d.e., clasei **BorcanCuGem** pot avea:
  - culoare și poziție diferită
  - diverse tipuri de gem în interior
- Astfel că, *variabilele instanță* ale lor au valori diferite
  - Notă: instanțele de obiect conțin variabile instanță — două moduri de folosire diferite, dar înrudite, a cuvântului *instanță*
- Instanțele individuale au identități individuale
  - aceasta permite altor obiecte să trimită mesaje unui obiect dat
  - fiecare instanță este unică, chiar dacă are aceleași capacități
    - gândiți-vă la clasa studenților de la acest curs



# Mesaje pentru comunicarea între obiecte

---

- Instanțele nu sunt izolate — ele trebuie să comunice cu altele pentru a-și realiza sarcina
  - proprietățile le permit să știe despre alte obiecte
- Instanțele trimit mesaje una alteia pentru a invoca o capabilitate (adică, pentru a executa o sarcină)
  - metoda reprezintă codul care implementează mesajul
  - spunem “apelează metoda” în loc de “invocă capabilitatea”



# Mesaje pentru comunicarea între obiecte

- Fiecare mesaj necesită:
  - *un emițător (expeditor)*: obiectul care inițiază acțiunea
  - *un receptor*: instanța a cărei metode este invocată
  - *numele mesajului*: numele metodei apelate
  - opțional *parametri*: informații suplimentare necesitate de metodă pentru a opera
    - mai multe mai târziu
- Receptorul poate (dar nu este nevoit) să trimită un răspuns
  - vom trata *tipurile returnate* în detaliu, mai târziu





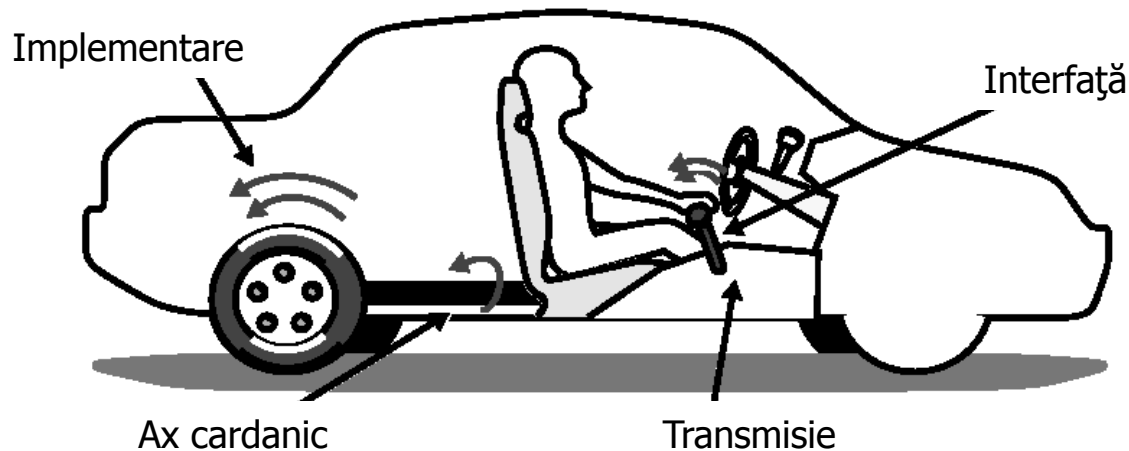
# Încapsulare

- Un automobil *încapsulează* multă informație
  - chiar literal, prin complexitatea construcției sale
- Dar nu este nevoie să știi cum funcționează o mașină pentru a o conduce
  - Volanul și schimbarea vitezelor constituie interfața
  - motorul, transmisia, axul cardanic, roțile, . . . , sunt implementarea (ascunsă)



# Încapsulare

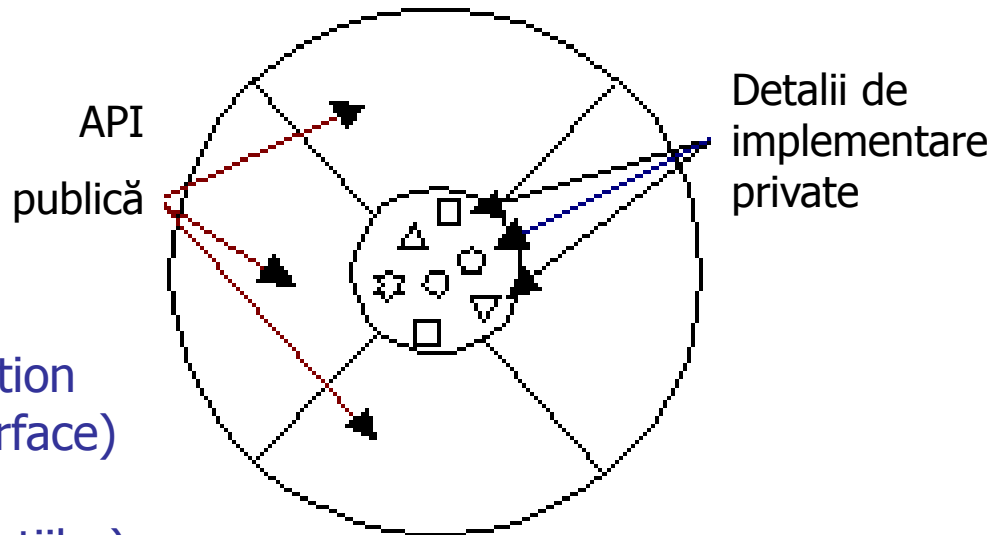
- Asemănător, nu e nevoie să știm cum funcționează un obiect pentru a-i trimite mesaje
- Dar, este nevoie să știm ce mesaje înțelege (adică, care îi sunt capabilitățile)
  - clasa instanței determină ce mesaje îi pot fi trimise





# Încapsularea

- Închiderea datelor într-un obiect
  - Datele nu pot fi accesate direct din afară
- Oferă securitatea datelor

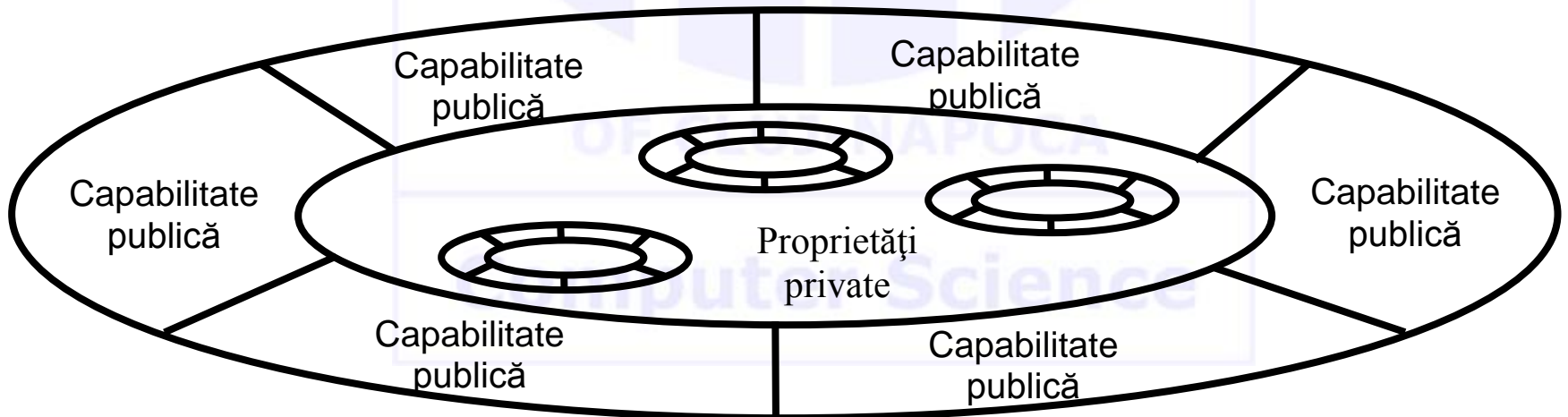


Notă. API (Application Programming Interface)  
= interfață pentru programarea aplicațiilor)



# Vederile unei clase

- Obiectele separă *interfața* de *implementare*
  - obiectul este "cutie neagră"; ascunde funcționarea și părțile interne
  - interfața protejează implementarea împotriva utilizării greșite





# Vederile unei clase

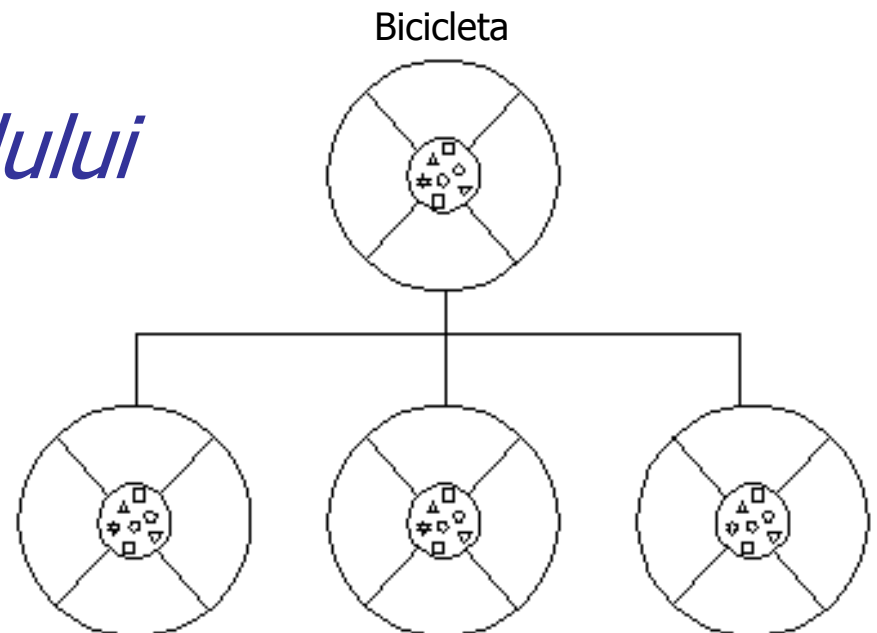
---

- Interfața: vedere *publică*
  - permite instanțelor să coopereze unele cu altele fără a ști prea multe detalii
  - ca un *contract*: constă dintr-o listă de capabilități și documentație pentru cum să fie folosite
- Implementarea: vedere *privată*
  - proprietățile care ajută capabilitățile să-și îndeplinească sarcinile



# Moștenirea

- O clasă (*subclasă*) poate *moșteni* attribute și metode dintr-o altă clasă (*superclasă*)
- Subclasele furnizează comportament specializat
- Oferă *reutilizarea codului*
- Evită *duplicarea* datelor





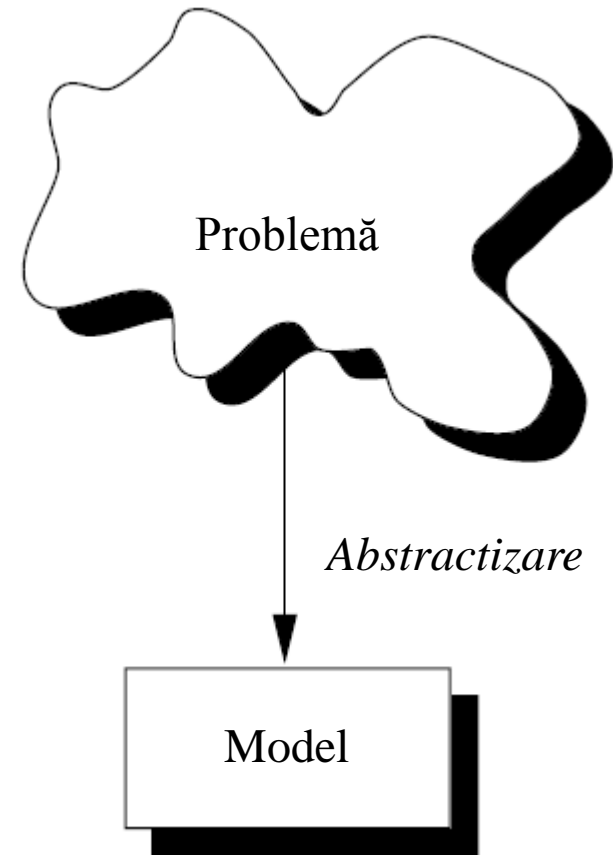
# Polimorfism

- Abilitatea de a lua multe forme
- *Aceeși metodă* folosită într-o superclasă poate fi *suprascrisă* în subclase pentru a da o *funcționalitate diferită*
- D.e. Superclasa 'Poligon' are o metodă numită, aflaSuprafata
  - aflaSuprafata în subclasa 'Triunghi'  $\rightarrow a=x*y/2$
  - aflaSuprafata in subclasa 'Dreptunghi'  $\rightarrow a=x*y$



# Abstractizare

- **Abstractizare:** eliminarea sau ascunderea deliberată a unor detalii ale unui proces sau artefact pentru a releva mai clar alte aspecte, detalii sau structura.





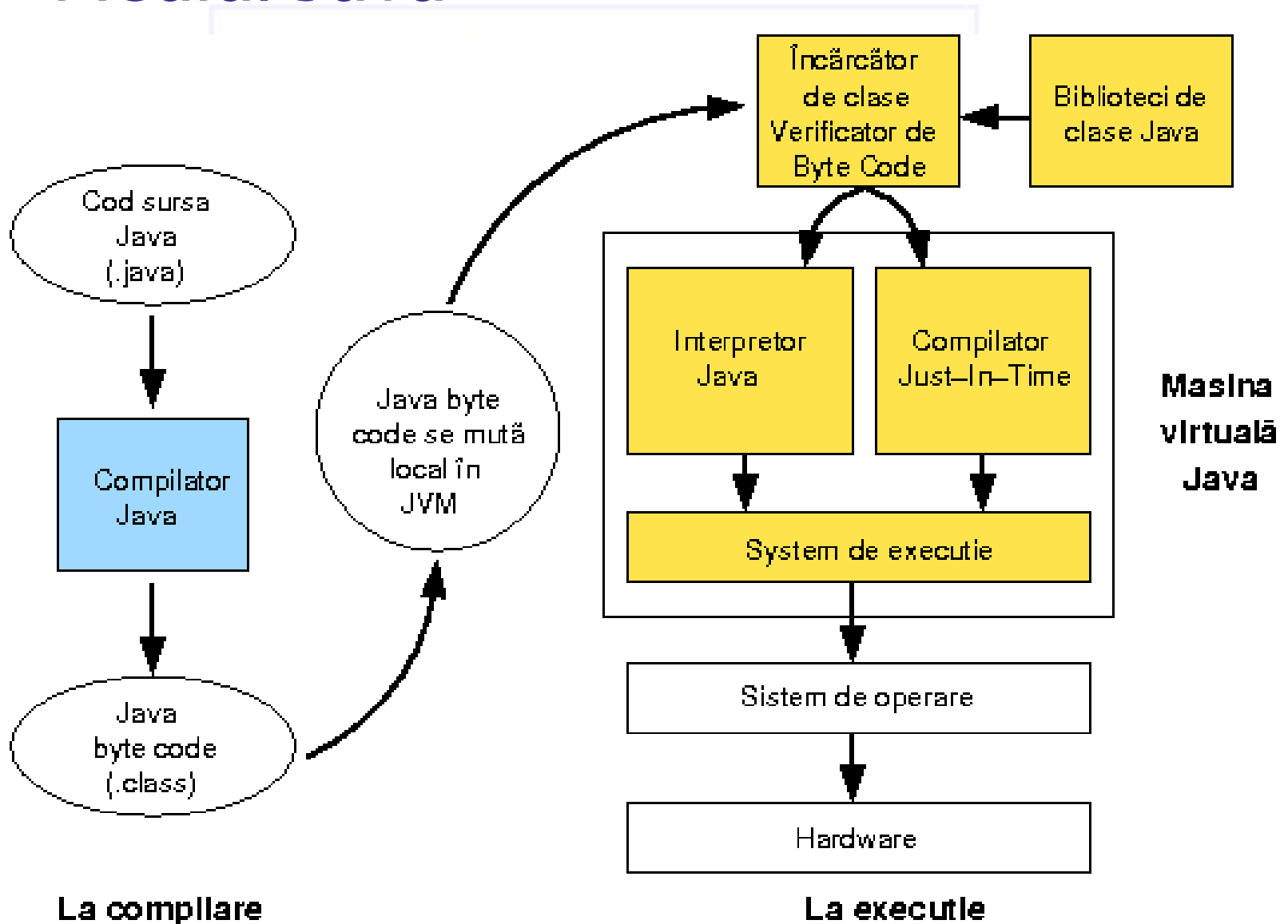


# Java. Caracteristici

- James Gosling și Patrick Naughton (conducătorii echipei care a dezvoltat limbajul), au definit limbajul Java ca fiind "Un limbaj simplu, orientat pe obiecte, care "înțelege" rețelele de calculatoare, interpretat, robust, sigur, neutru față de arhitecturi, portabil, de înaltă performanță, multi-fir, dinamic".



# Mediul Java





# Executarea programelor Java

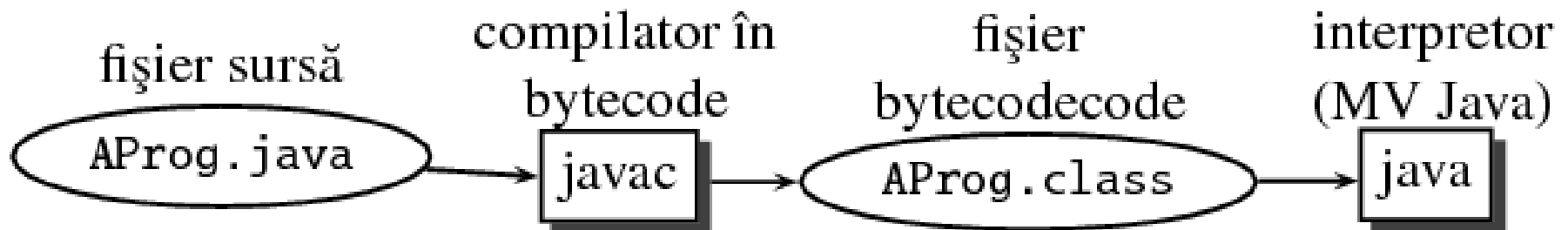
---

- Java folosește un proces în doi pași
- Compilează programul în *bytecodes*
  - bytecode este apropiat de formatul instrucțiunilor în limbaj mașină, dar nu chiar la fel — este un "limbaj mașină" a generic
  - nu corespunde nici unui procesor real
- *Mașina virtuală* (VM) interpretează bytecode în limbaj mașină nativ și-l rulează
  - există mașini virtuale diferite pentru calculatoare diferite, deoarece bytecode nu corespunde unei mașini reale



# Executarea programelor Java

- Se utilizează *același* bytecode Java pe *calculatoare diferite* fără a recompila codul sursă
  - fiecare VM interpretează același bytecode Java
  - aceasta permite să se execute programe Java prin simpla obținere a bytecodes din pagini de Web
- Aceasta face codul Java să ruleze **peste-platforme**
  - marketing-ul spune, "Scrie o dată, rulează oriunde!"
  - adevărat pentru "Java pur", nu pentru variante





# Aplicații Java

- Tipuri de aplicații Java:
  - *aplicații de-sine-stătătoare* și *applets/servlets*
- Un *aplicație de-sine-stătătoare* sau un program "obișnuit" este o clasă care are o metodă numită **main**
  - Când se lansează programul Java respectiv, *sistemul de execuție* invocă automat metoda numită **main**
  - Toate *aplicațiile de-sine-stătătoare* încep din metoda **main**
- Deja ați văzut asemenea aplicații în exemplele BlueJ



# Applets

- Un *applet* Java (o aplicație având caracteristici limitate, care necesită resurse de memorie limitate și portabilă între sistemele de operare) este un program Java destinat a fi rulat dintr-un browser de Web
  - poate fi rulat dintr-o locație de pe Internet
  - poate fi rulată și din vizualizatorul de applet-uri (**appletviewer**) pentru depanare
  - Applet-urile folosesc întotdeauna o interfață cu ferestre
- Aplicațiile de sine stătătoare pot folosi fie interfața cu ferestre sau I/O consolă (adică în modul text)
- Deja ați văzut un applet în exemplele BlueJ



# Încărcătorul de clase

- Programele Java sunt divizate în unități mai mici numite *clase*
  - Fiecare definiție de clasă este în mod normal într-un fișier separat și compilată separat
- *Încărcătorul de clase*: un program care leagă bytecod-ul claselor necesare pentru a rula un program Java
  - În alte limbaje de programare, corespondentul său este *editorul de legături (linkeditor)*



## Cuvinte cheie Java (cuvinte rezervate)

---

- Cuvinte care nu pot fi folosite la altceva decât în modul predefinit din limbaj
  - *abstract, assert, boolean, break, byte, case, catch, char, class, const, continue, default, do, double, else, extends, final, finally, float, for, goto, if, implements, import, instanceof, int, interface, long, native, new, package, private, protected, public, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, try, void, volatile, while*
  - *null, true, false* – predefinite ca literali





# Compilarea unui program sau a unei clase Java

- Fiecare definiție de clasă trebuie să se afle într-un fișier al cărui nume este numele clasei urmat de extensia **.java**
  - Exemplu: Clasa **UnProgram** trebuie să se afle în fișierul numit **UnProgram.java**
- Fiecare clasă este compilată folosind comanda **javac** urmată de numele fișierului care conține clasa

```
javac UnProgram.java
```

- Rezultatul este un program în byte-code cu același nume ca al clasei, urmat de extensia **.class**

```
UnProgram.class
```



# Rularea unui program Java

- Un program Java poate fi *rulat* (**java**) după ce i-au fost compilate toate clasele
  - Rulați doar clasa care conține o metodă **main** (sistemul va încărca și rula celelalte clase automat, dacă mai sunt)
  - Metoda **main** începe cu linia:  
`public static void main(String[ ] args)`
  - Comanda de lansare trebuie urmată doar de numele clasei (fără extensii)  
`java UnProgram`
- Vedeți **Hello.java** din exemplele BlueJ



# Convenții pentru nume

- Începeți numele de variabile, metode și obiecte cu o literă mică, indicați limitele "cuvintelor" cu o litera mare și pentru celelalte caractere folosiți doar litere și cifre ("camelcase")

`vitezaMaxima`      `rataDobinzii`      `oraSosirii`

- Începeți numele de clase cu majusculă și pentru restul identificatorului aplicați regula de mai sus

`UnProgram`      `OClasa`      `String`



# Declararea variabilelor

- Fiecare variabilă dintr-un program Java trebuie *declarată* înainte de utilizare
  - Declarația informează compilatorul asupra tipului de dată care va fi stocat în variabilă
  - Tipul variabilei este urmat de unul sau mai multe nume separate de virgule și terminat cu punct și virgulă
  - Variabilele se declară de obicei chiar înainte de folosire sau la începutul unui bloc (indicat de o acoladă deschisă { )
  - Tipurile simple în Java sunt numite *tipuri primitive*  
`int numarulDeCai;`  
`double oLungime, lungimeaTotala;`



# Modificatori de acces pentru variabile/metode

## ■ **private**

- variabila/metoda este vizibilă local în cadrul clasei.
- *"Vizibilă doar mie"*.

## ■ **protected**

- variabila/metoda poate fi văzută din toate clasele, subclasele și celelalte clase din același pachet (package).
- *"Vizibilă în familie"*.

## ■ **public**

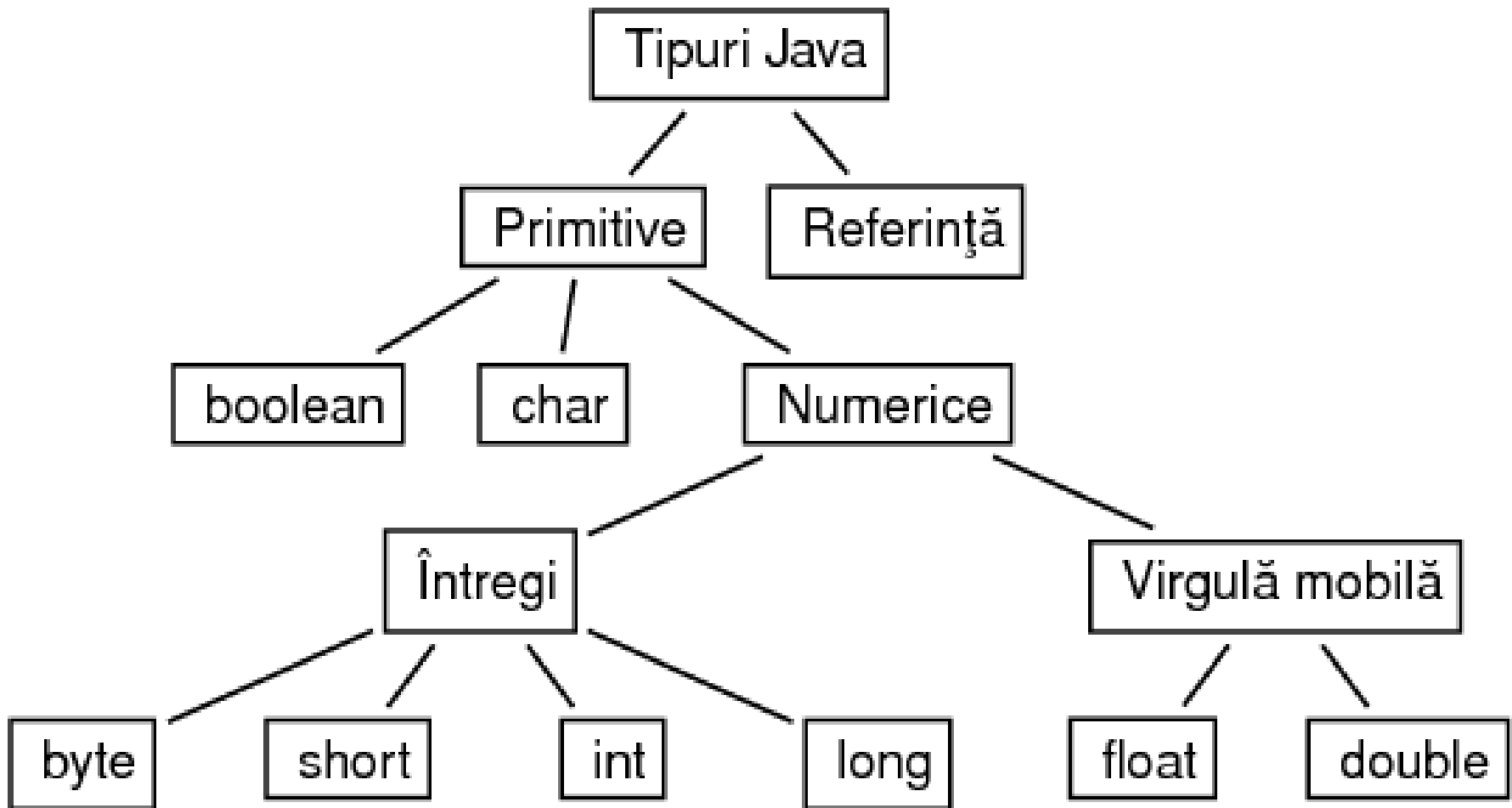
- variabila/metoda poate fi văzută din toate clasele
- *"Vizibilă tuturor"*.

## ■ Modificatorul de acces implicit, nu are cuvânt cheie.

- **public** pentru ceilalți membrii din același pachet.
- **private** pentru oricine din afara pachetului
- numită și *acces în pachet*.
- *"Vizibilă în vecinătate"*



# Tipuri Java





# Tipuri primitive

Tip primitiv	Biți	Minimum	Maximum	Wrapper type
<b>boolean</b>	—	—	—	<b>Boolean</b>
<b>char</b>	16-bit	Unicode 0	Unicode $2^{16}-1$	<b>Character</b>
<b>byte</b>	8-bit	-128	+127	<b>Byte</b>
<b>short</b>	16-bit	$-2^{15}$	$+2^{15}-1$	<b>Short</b>
<b>int</b>	32-bit	$-2^{31}$	$+2^{31}-1$	<b>Integer</b>
<b>long</b>	64-bit	$-2^{63}$	$+2^{63}-1$	<b>Long</b>
<b>float</b>	32-bit	IEEE754	IEEE754	<b>Float</b>
<b>double</b>	64-bit	IEEE754	IEEE754	<b>Double</b>
<b>void</b>	—	—	—	<b>Void</b>

Toate tipurile numerice sunt cu semn.



# Compatibilitate la asignare

- Mai general, o valoare de orice tip din lista următoare poate fi asignată unei variabile de orice alt tip care apare la dreapta ei

`byte` → `short` → `int` → `long` → `float` → `double`  
`char` \_\_\_\_\_ ↗

- Gama valorilor de la dreapta este mai largă
- Este necesară o conversie de tip explicită (*type cast*) pentru a asigna o valoare de un tip la o variabilă care apare la stânga ei în lista de mai sus (d.e., `double` la `int`)
- Observați că în Java un `int` nu poate fi asignat la o variabilă de tip `boolean`, nici un `boolean` la o variabilă de tip `int`





# Operatori aritmetici și expresii

- Ca în majoritatea limbajelor, și în Java se pot forma *expresii* folosind variabile, constante și operatori aritmetici
  - Operatori aritmetici;  $+$  (adunare),  $-$  (scădere),  $*$  (înmulțire),  $/$  (împărțire),  $\%$  (modulo, rest)
  - Se poate folosi o expresie oriunde este legal să se folosească o valoare de tipul produs de expresie



# Operatori aritmetici și expresii

- Dacă se combină un operator aritmetic cu operanzi de tipul **int**, atunci tipul rezultat este **int**
- Dacă se combină un operator aritmetic cu unul sau doi operanzi de tipul **double**, atunci tipul rezultat este **double**
- La combinarea de operanzi de tip diferit, tipul rezultat este cel mai din dreapta din lista de mai jos care se află în expresie

**byte** → **short** → **int** → **long** → **float** → **double**  
**char**      \_\_\_\_\_ ↗

- Excepție: Dacă tipul rezultat este **byte** sau **short** (potrivit regulii date), atunci tipul produs va fi de fapt un **int**



## Reguli de precedență și asociativitate

---

- La determinarea ordinii operațiilor adiacente, operația cu precedență mai mare (și argumentele sale aparente) este grupată înaintea operației de precedență mai mică  
$$\text{base} + \text{rate} * \text{hours} \text{ se evaluează ca}$$
$$\text{base} + (\text{rate} * \text{hours})$$
- La precedență egală, ordinea operațiilor este determinată de regulile de *asociativitate*



# Reguli de precedență și asociativitate

- Operatorii *unari* de precedență egală sunt grupați de la dreapta la stânga  
`++rate` se evaluează ca `+( -(+rate) )`
- Operatorii *binari* de precedență egală sunt grupați de la stânga la dreapta  
`base + rate + hours` se evaluează ca  
`(base + rate) + hours`
- Excepție: Un șir de operatori de asignare este grupat de la dreapta la stânga  
`n1 = n2 = n3;` se evaluează ca `n1 = (n2 = n3);`



# Posibilă problemă: Erorile de rotunjire la numerele în virgulă mobilă

---

- Numerele în virgulă mobilă sunt, în general, doar valori aproximative
  - Matematic, numărul în virgulă mobilă  $1.0/3.0$  este egal cu  $0.33333333 \dots$
  - Un calculator are o cantitate limitată de memorie
    - Poate stoca  $1.0/3.0$  ca ceva în genul lui  $0.3333333333$ , puțin mai puțin decât o treime
  - De fapt numerele sunt stocate binar, dar consecințele sunt aceleași: numerele în virgulă mobilă pot pierde precizie



# Împărțirea întreagă și cea în virgulă mobilă

- Dacă unul sau amândoi operanzii sunt în virgulă mobilă, împărțirea dă un rezultat în virgulă mobilă  
 $15.0/2$  se evaluează la  $7.5$
- Cum ambii operanzi întregi, împărțirea dă un întreg
  - O eventuală parte fracționară este ignorată
  - Nu se fac rotunjiri  
 $15/2$  se evaluează la  $7$
- Aveți grijă ca cel puțin un operand să fie în virgulă mobilă dacă este nevoie de partea fracționară



# Conversia de tip explicită

- O *conversie de tip explicită (type cast)* ia o valoare de un tip și produce o valoare "echivalentă" de celălalt tip
  - Dacă **n** și **m** sunt întregii de împărțit și e nevoie de partea fracționară, atunci cel puțin un operand trebuie să fie în virgulă mobilă **înainte** de efectuarea operației  
`double ans = n / (double)m;`
  - La fel ca în C, tipul dorit este pus între paranteze imediat înaintea variabilei de convertit
  - Tipul și valoarea variabilei de convertit nu se schimbă



# Conversia de tip explicită

- La conversia explicită de la virgulă mobilă la întreg, numărul este trunchiat, nu rotunjit
  - `(int)2.9` se evaluează la `2`, nu `3`
- La asignarea valorii unui întreg la o variabilă în virgulă mobilă, Java realizează o conversie explicită de tip automată numită *coerciție de tip*
- Nu este legal să se atribuie un `double` la un `int` fără o conversie explicită

```
int i = 5.5; // Ilegal
```

```
int i = (int)5.5 // Corect
```





# Operatorii increment și decrement

- Când oricare dintre operatorii `++` sau `--` precede o variabilă și este o parte a expresiei, expresia este evaluată folosind valoarea modificată a variabilei
  - Dacă `n` este `2`, atunci `2 * (++n)` se evaluează la `6`
- Când oricare dintre operatori urmează unei variabile și este parte a expresiei, expresia este evaluată folosind valoarea originală și abia apoi se schimbă valoarea variabilei
  - Dacă `n` este `2`, atunci `2 * (n++)` se evaluează la `4`



# Concepte prezentate

- Paradigme în programare
  - structurată
  - orientată pe obiecte
- Abstractizare
  - tip de dată abstract
- Obiect
  - capabilități(comportamente): acțiuni
    - constructor
    - comenzi
    - interogări
  - proprietăți: stare
    - attribute
    - componente
    - asocieri
  - instanță a unei clase
- Comunicarea între obiecte: mesaje
- Încapsulare
- Clasă
  - șablon pentru crearea obiectelor
  - capabilitățile obiectelor: implementate ca metode
  - proprietățile obiectelor: variabile instanță
  - interfață – vedere publică
  - implementare – vedere privată
- Moștenire
  - superclasă, subclasă
- Polimorfism