



TECHNICAL UNIVERSITY

OF CLUJ-NAPOCA

Computer Science

# Programare orientată pe obiecte

1. Despre curs
2. Concepte și paradigme în POO
3. Mediul Java
4. Variabile și tipuri
5. Operatori aritmetici și expresii



# Cadre didactice

## ■ Curs

- Ion Giosan
  - E-mail: [Ion.Giosan@cs.utcluj.ro](mailto:Ion.Giosan@cs.utcluj.ro)
  - Web page:  
[http://users.utcluj.ro/~igiosan/teaching\\_poo.html](http://users.utcluj.ro/~igiosan/teaching_poo.html)

## ■ Laborator

- Claudiu Toader (grupa 6)
- George Cucoş-Artene (grupa 7)
- Cristina Rădulescu (grupa 8)
- Sergiu Nicula (grupa 9)
- Bogdan Potra (grupa 10)



# Conținutul cursului

- 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. Pachete
- Moștenirea și polimorfismul
- Interfețe Java
- Reprezentarea în UML a claselor și legăturilor dintre ele
- Excepții și tratarea lor
- Colecții Java
- Interfețe utilizator (*GUIs*) în Java
- Testarea și depanarea programelor
- Introducere în sistemul I/E Java
- Fire de lucru (*threads*) în Java



# Evaluare. Referințe

## ■ Evaluare

- Nota finală =  $0.4 * \text{Laborator} + 0.5 * \text{ExamenScris} + 0.1 * \text{TesteCurs}$ 
  - Obligatoriu Laborator  $\geq 5$
  - Obligatoriu ExamenScris  $\geq 5$
- Bonusuri
  - Prezență sporită la cursuri

## ■ Referințe

- **Bruce Eckel, *Thinking in Java*, 4th edition, Prentice Hall, 2006**
- **Kathy Sierra, Bert Bates, *SCJP Sun Certified Programmer for Java 6*, Mc Graw Hill, 2008**
- Tutoriale Java și Documentația Java de la Oracle
- Tutoriale UML introductory
- Documentația Eclipse



# Paradigme de programare

## ■ Paradigmă de programare

- Un model care descrie esența și structura computației
- Un stil fundamental de a programa
- Oferă și determină viziunea pe care o are programatorul asupra execuției programului

Exemple:

- În programarea funcțională un program poate fi conceput ca fiind o secvență de evaluări de funcții, fără stări
- În POO, programatorii pot concepe programele ca fiind o colecție de obiecte care interacționează

Computer Science



# Paradigme de programare

Programarea OO

Abstractizarea datelor

Programarea structurată

Programarea imperativă



# Programarea imperativă

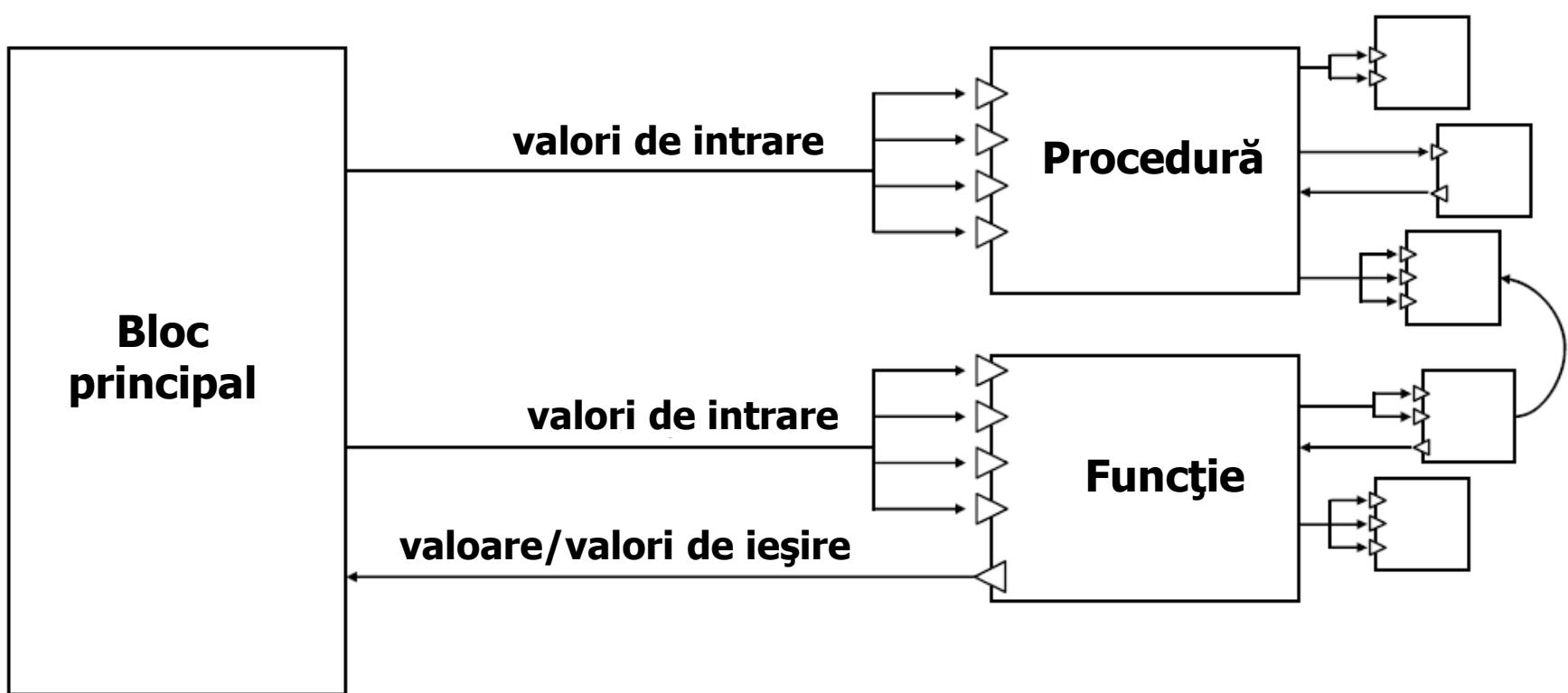
- Un calculator în modelul tradițional von Neumann
  - Unitate centrală de prelucrare
  - Memorie
  - Efectuează secvențe de instrucțiuni atomice care accesează, operează asupra valorilor stocate la locații de memorie adresabile individual și le modifică
- O computație este o serie de operații aritmetice și de efecte laterale, cum sunt atribuirile sau transferurile de date care modifică starea unității de stocare, intrarea sau ieșirea
- Este de subliniat importanța atribuirilor și a variabilelor pe post de containere pentru paradigma imperativă
- Exemple de limbaje: Fortran, Pascal, C, Ada.

Computer Science



# Programarea structurată

Program procedural = definiții de date și apeluri de funcții/proceduri





# Programarea structurată

## ■ Abstractizarea operațiilor

### ■ Structura unui modul

#### ■ Interfață

- 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



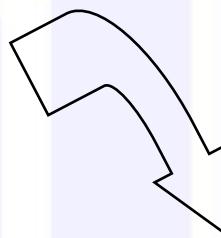
# Beneficiile programării structurate

- 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

- Impunerea unei separări clare între *proprietățile abstracte* ale unui *tip de dată* și *detaliiile 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 încorpora 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 domeniul real 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 orientării pe obiecte

- Paradigma programării structurate a avut inițial succes (1975-85)
  - Dar a început să eșueze la produse mai mari (> 50,000 LOC)
  - 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
    - orientate pe atrbute
    - ...**dar nu amândouă**



# Paradigma orientării pe obiecte

- O simulare a domeniului unei probleme prin abstractizarea informațiilor de comportament și stare din obiecte din lumea reală
- Conceptele POO
  - Clase și Obiecte
  - Abstractizare și Încapsulare
  - Transmitere de mesaje
  - Moștenire și Polimorfism



# 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 incorporează 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



# Paradigma orientării pe obiecte

- Totul este reprezentat prin obiecte  
(obiect = o variabilă mai specială ce încapsulează atât date cât și operații cu aceste date)
- Obiectele comunică între ele prin trimitere/ primire de mesaje (mesaj = apel de metodă)
- Obiectele au propria lor memorie
- Orice obiect are un tip  $\Leftrightarrow$  orice obiect e o instanță a unei clase (unde 'clasa' este sinonim cu 'tip')
- Toate obiectele de un anumit tip pot trimite sau primi aceleasi mesaje



# Punctele tari ale POO

- Ascunderea informației => întreținerea post-livrare este mai sigură
  - Sansele 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 POO)

Computer Science



# Punctele tari ale POO

- 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 independentă promovează reutilizarea codului



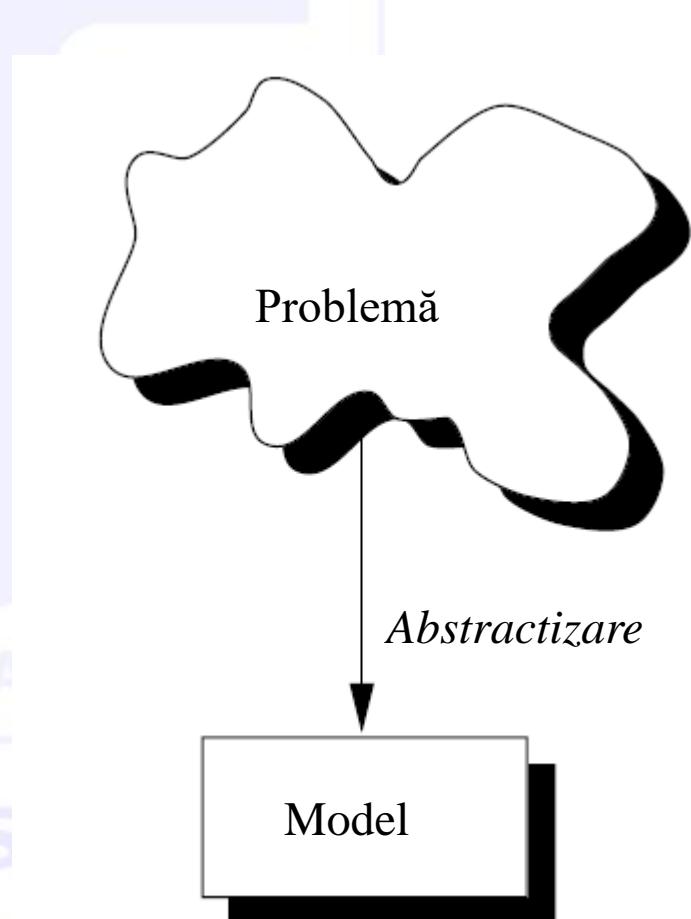
# Programarea orientată pe obiecte

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



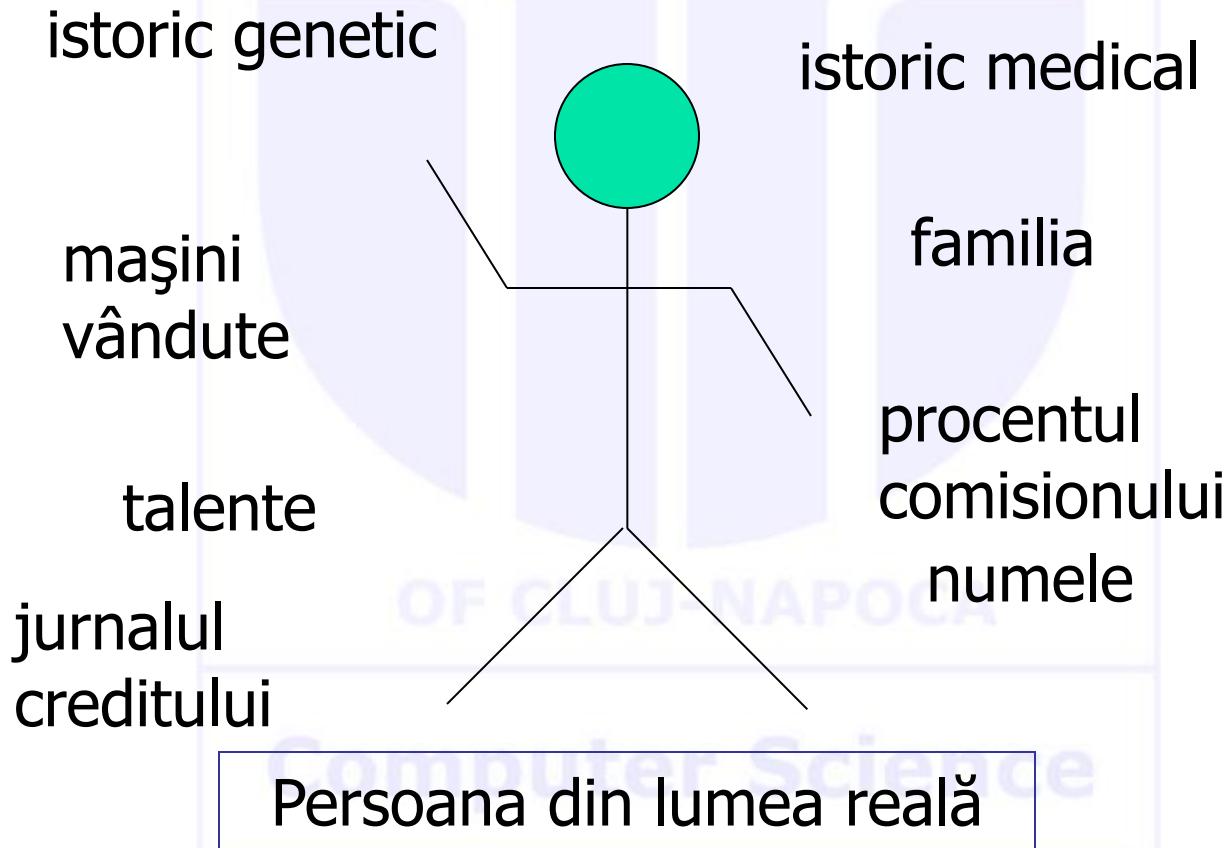
# Concepte POO

- *Abstractizare* = proces de filtrare a detaliilor neimportante ale obiectului astfel încât să rămână doar caracteristicile importante
- Ne ocupăm doar de datele care prezintă interes pentru problema noastră



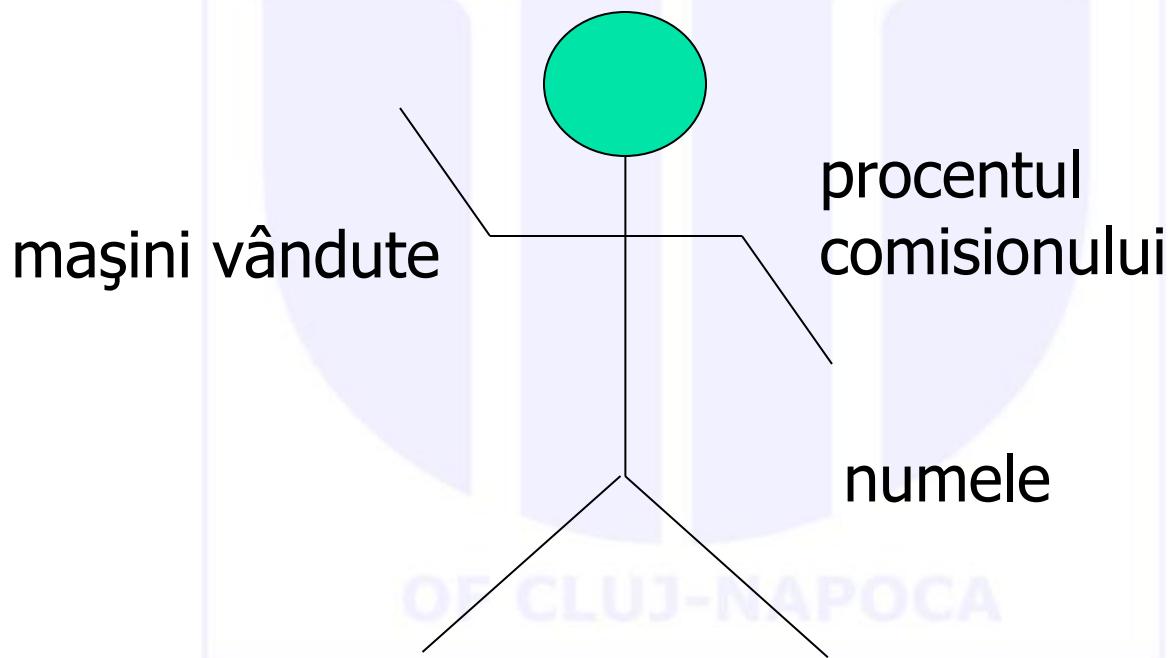


# Abstractizare. Exemplu





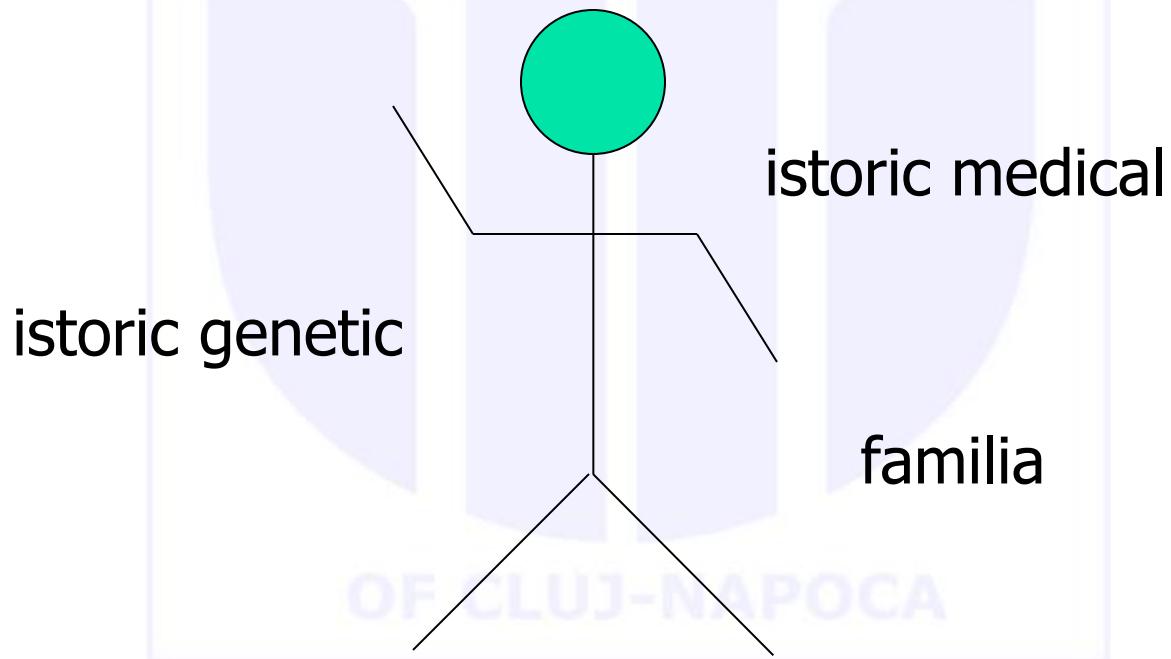
# Abstractizare. Exemplu



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



# Abstractizare. Exemplu



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-si "mesaje" unul altuia
- Obiectele modelează lucruri *tangibile*
  - Persoană
  - Bicicletă
  - Cal
  - Bancă etc.



# Ce sunt obiectele software?

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



# Proprietățile obiectului: starea

- *Proprietățile* : determină cum acționează un obiect
  - Pot fi constante (nu se schimbă) sau variabile
  - Pot fi ele însăși obiecte — pot primi mesaje
  - Ex. Capacul *borcanului cu gem* și gemul în sine sunt obiecte
- Proprietățile pot fi:
  - **atribute**: 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 acestuia



# 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*
  - *atribute*: culoare, material, miros
  - *componente*: capac, container, etichetă
  - *asocieri*: un *borcan cu gem* poate fi asociat cu încăperea în care se află



# 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
  - *acțiuni*: modifică proprietățile obiectului
  - *interrogări*: furnizează răspunsuri bazate pe proprietățile obiectului



# Capabilitățile obiectelor: acțiuni

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



# 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 multimi 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 multimi de obiecte individuale
  - toate *borcanele cu gem* se pot deschide, încide și goli
- definește un şablon 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



# Instante de obiecte

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



# Instante de obiecte

- Instante 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: instantele de obiect conțin variabile instanță — două moduri de folosire diferite, dar înrudite, a cuvântului *instanță*
- Instantele individuale au identități individuale
  - aceasta permite altor obiecte să trimită mesaje unui obiect dat
  - fiecare instanță este unică, chiar dacă are aceleași capabilități
    - Exemplu: 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 capacitate (adică, pentru a executa o sarcină)
  - metoda reprezintă codul care implementează mesajul
  - spunem “apeleză metoda” în loc de “invocă capacitatea”



# 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
  - optional *parametri*: informații suplimentare necesitate de metodă pentru a opera
- Receptorul poate (dar nu este nevoie) să trimită un răspuns
  - prin *tipurile returnate*



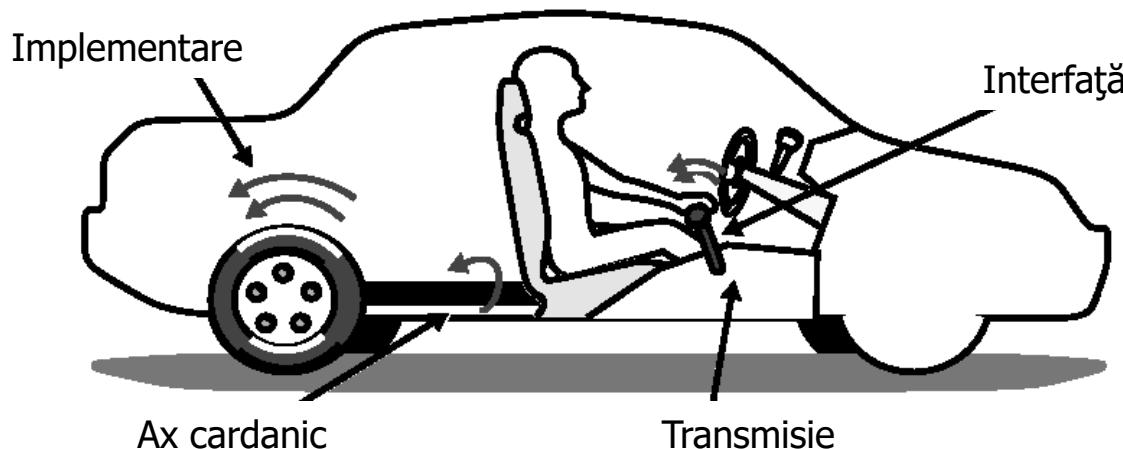
# Încapsulare

- 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

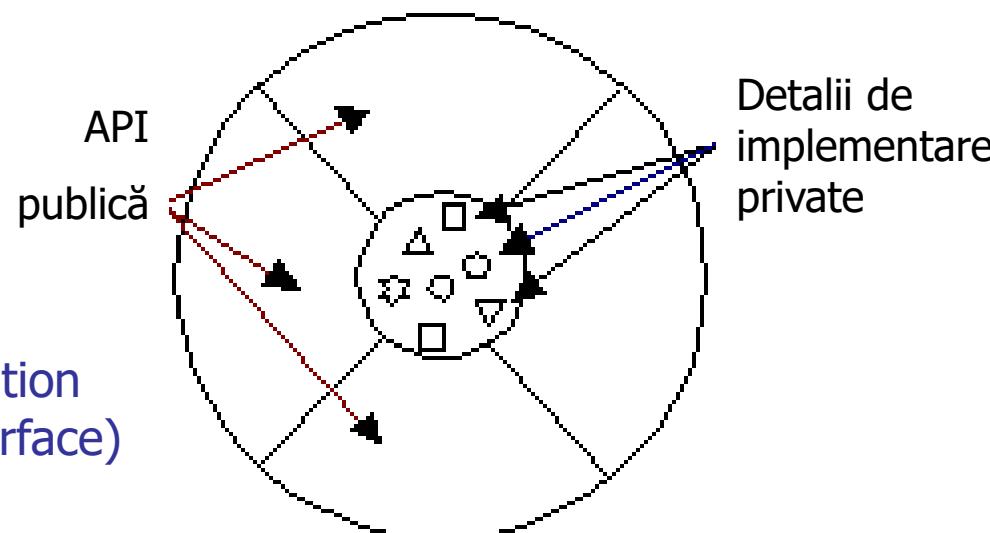
- Asemănător, nu e nevoie să știm cum funcționează un obiect pentru a-i trimite mesaje
- Dar, este nevoie să știm ce mesaje înțelege (adică, care îi sunt capabilitățile)
  - clasa instanței determină ce mesaje îi pot fi trimise





# Încapsulare

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



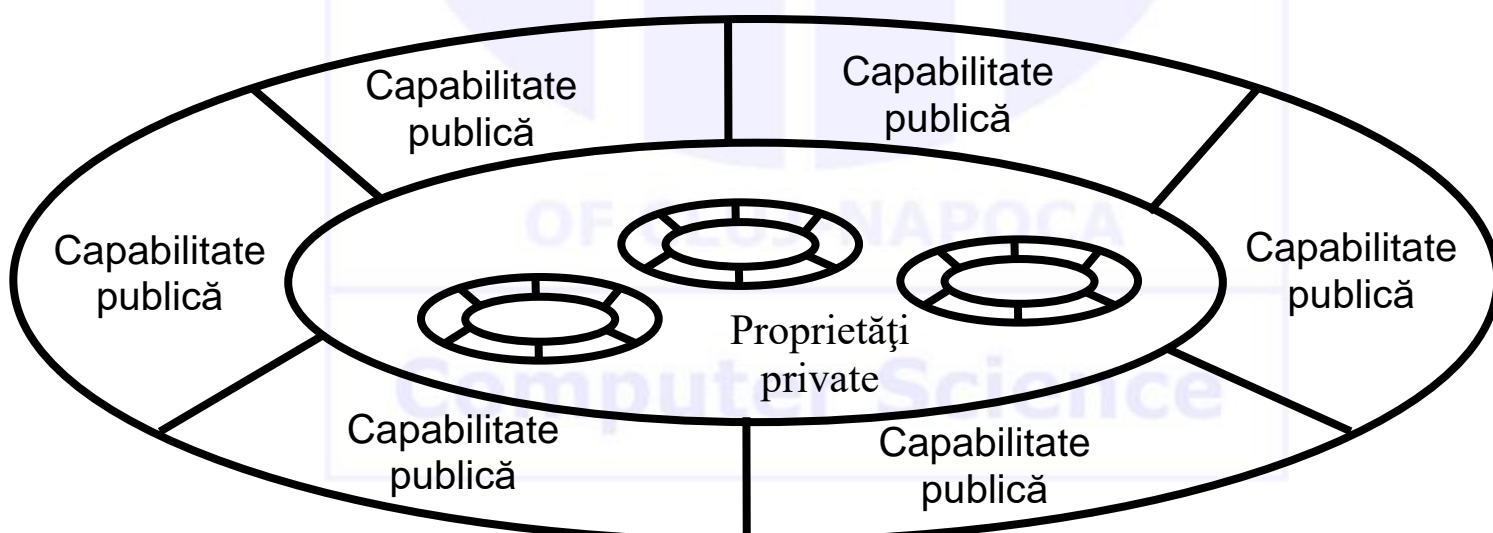
Notă. API (Application Programming Interface) = interfață pentru programarea aplicațiilor



# Vederile unei clase

## ■ Obiectele separă *interfața* de *implementare*

- obiectul este “cutie neagră”; ascunde funcționarea și părțile interne
- interfața protejează implementarea împotriva utilizării greșite





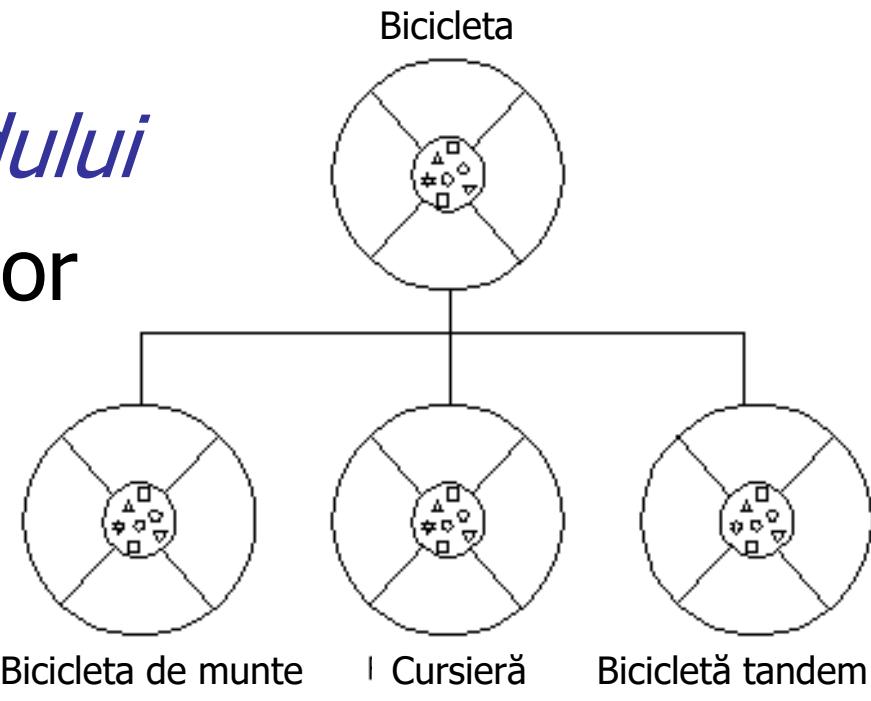
# Vederile unei clase

- 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
- *Aceeași 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' →  $a=x*y/2$
  - aflaSuprafata* în subclasa 'Dreptunghi' →  $a=x*y$



# Java. Caracteristici

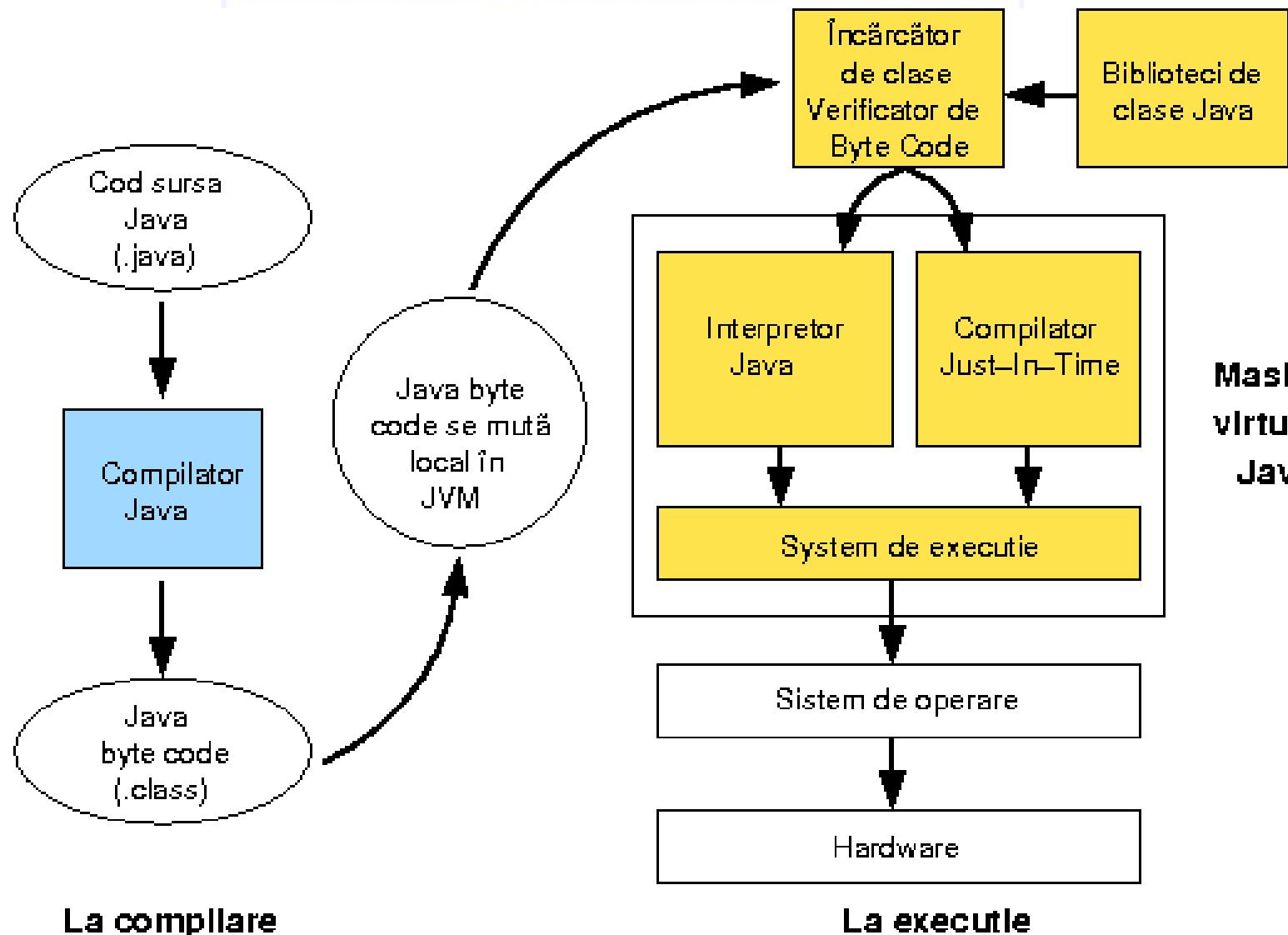
- James Gosling și Patrick Naughton (conducătorii echipei care a dezvoltat limbajul), au definit limbajul Java ca fiind

"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"

Computer Science



# Mediul Java



**La compilare**

**La execuție**



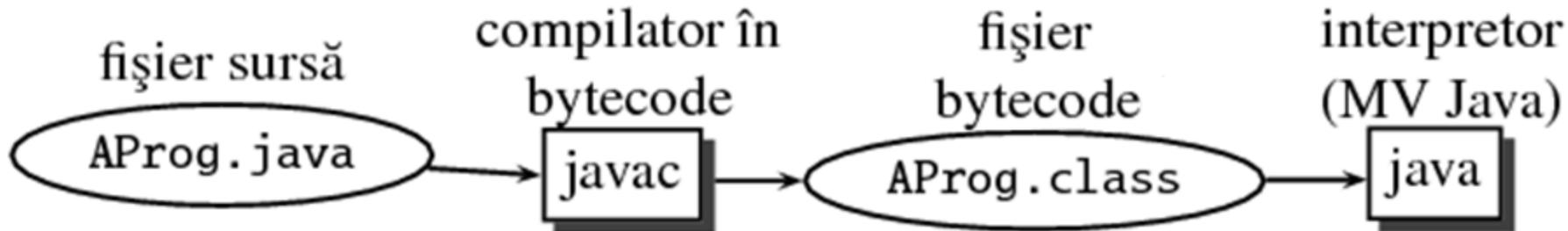
# Executarea programelor Java

- 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" 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 recompile 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-stătoare*
  - *applets/servlets*
- O *aplicație de-sine-stătoare* 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-stătoare* încep din metoda **main**



# Applet-uri vs. Aplicații de sine stătătoare

- Un *applet* Java este un program Java destinat a fi rulat dintr-un browser de Web
  - Applet-urile folosesc întotdeauna o interfață cu ferestre
  - Este o aplicație având caracteristici limitate, care necesită resurse de memorie limitate și portabilă între sistemele de operare
- Aplicațiile *de-sine-stătătoare* pot folosi atât interfața cu ferestre cât și I/E de consolă (adică în modul text)

Computer Science



# 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* este un program care leagă *bytecode*-ul claselor necesare pentru a rula un program Java
  - În alte limbi de programare, corespondentul său este *editorul de legături (link-editor)*



## 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** are semnătura

```
public static void main(String[ ] args)
```
  - Comanda de lansare a programului trebuie urmată doar de numele clasei (fără extensii)  
**java UnProgram**



# Convenții pentru nume

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

`vitezaMaxima`    `rataDobanzii`    `oraSosirii`

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

`UnProgram`    `OClasa`    `String`

`Computer Science`



# Declararea variabilelor

- Fiecare variabilă dintr-un program Java trebuie *declarată* înainte de utilizare
  - Declarația informează compilatorul asupra tipului de date 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**

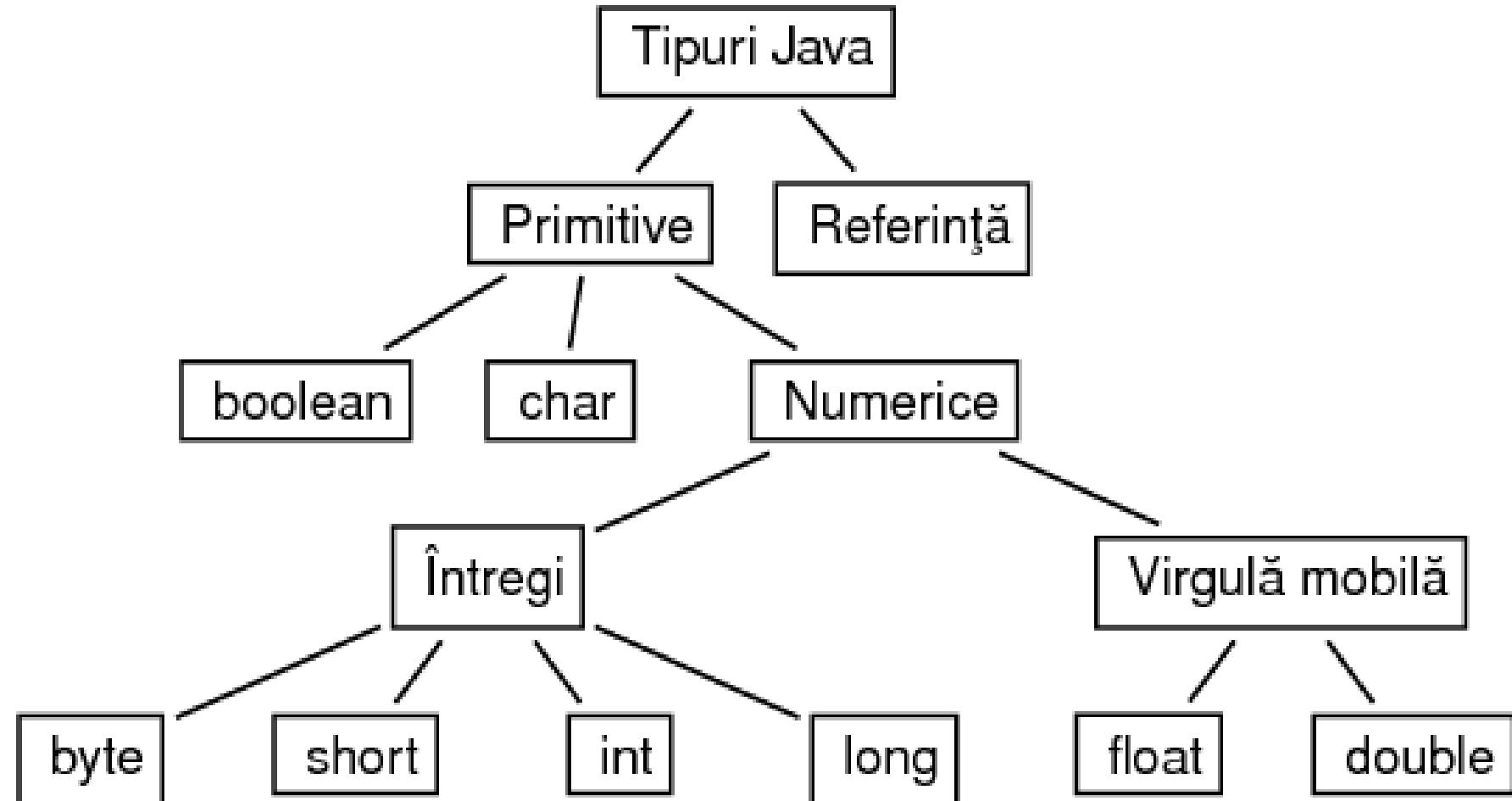
- variabila/metoda poate fi văzută din toate clasele
- "*Vizibilă tuturor*"

## ■ 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**  

  - 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**

- Exceptie: 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
  - base + rate \* hours** se evaluează ca  
**base + (rate \* hours)**
- La precedență egală, ordinea operațiilor este determinată de regulile de *asociativitate*

Computer Science



# 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  $0.3333333 \dots$
  - 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 operanze 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 fractionară 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 fractionară



# 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*
- `double d = 5;`
- Nu este legal să se atribuie un `double` la un `int` fără o conversie explicită

```
int i = 5.5; // Illegal  
int i = (int)5.5 // Correct
```



# 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 evaluatează 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 evaluatează la **4**



TECHNICAL UNIVERSITY

Computer Science

# Programare orientată pe obiecte

1. Clase învelitoare (*wrapper classes*)
2. Câteva observații despre operatori
3. Structuri de control în Java
4. Clase și Obiecte



# Clase Învelitoare

- În POO, clasele învelitoare sunt clasele care încapsulează tipurile primitive, astfel încât să se poată manipula ca niște obiecte și pentru a le putea 'ataşa' metode (în special cu rol de conversie)

<b>Tipul primitiv</b>	<b>Clasa Învelitoare</b>	<b>Argumentele constructorilor</b>
byte	Byte	byte sau String
short	Short	short sau String
int	Integer BigInteger	int sau String String
long	Long	long sau String
float	Float	float, double sau String
double	Double BigDecimal	double sau String String
char	Character	char
boolean	Boolean	boolean sau String



# Clase Învelitoare

- *Boxing* (împachetare): procesul de trecere de la o valoare primitivă la un obiect al clasei sale învelitoare
  - Pentru a converti valoarea la una "echivalentă" de tip clasă, creați un obiect de tipul corespunzător folosind valoarea primitivă ca argument
  - Noul obiect va conține o variabilă instanță care stochează o copie a valorii primitive
  - Spre deosebire de alte clase, clasele învelitoare nu au constructori fără argumente

```
Integer integerObject = new Integer(42);
```



# Clase Învelitoare

- *Unboxing* (despachetare): procesul de trecere de la un obiect al unei clase învelitoare la valoarea corespunzătoare a unui tip primitiv
  - Metodele de convertire a unui obiect din clasele **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, și **Character** la valorile primitive corespunzătoare sunt (în ordine) **byteValue**, **shortValue**, **intValue**, **longValue**, **floatValue**, **doubleValue**, și **charValue**
  - Nici una dintre aceste metode nu necesită argumente

```
int i = integerObject.intValue();
```

Computer Science



# Boxing și Unboxing automat

- Începând cu versiunea 5.0, Java poate face automat împachetare/despachetare
- În loc să creăm un obiect din clasa învelitoare (ca mai înainte), se poate face o conversie de tip automată
  - `Integer integerObject = 42;`**
    - Java nu garantează că două referințe de astfel de obiecte care împachetează automat o aceeași valoare primitivă sunt diferite!
- În loc să trebuiască să invocăm metoda corespunzătoare (ca **`intValue`**, **`doubleValue`**, **`charValue`** etc.) pentru a converti un obiect din clasa învelitoare la tipul său asociat, valoarea primitivă poate fi recuperată automat
  - `int i = integerObject;`**



# Boxing == și *equals()*

- Pentru a verifica dacă două variabile de tip referință au aceeași valoare, vom folosi metoda *equals()*
- Exemplu:

```
Integer i1 = 42; //împachetare automată
Integer i2 = 42; //împachetare automată
if (i1 != i2) // nu este garantat că sunt diferite!
    System.out.println("Obiecte diferite");
if (i1.equals(i2)) // garantat că sunt identice!
    System.out.println("Variabile cu valori egale");
```

## Afișare la ieșirea standard:

Obiecte diferite (nu este garantat că se afișează!)

Variabile cu valori egale



# Constante și metode statice în clasele învelitoare

- Constante pentru valori maxime și minime pentru tipurile primitive
  - Exemple: `Integer.MAX_VALUE`, `Integer.MIN_VALUE`,  
`Double.MAX_VALUE`, `Double.MIN_VALUE` etc.
- Clasa `Boolean` are nume pentru două constante de tipul `Boolean`
  - `Boolean.TRUE` și `Boolean.FALSE` sunt obiectele booleene corespunzătoare valorilor `true` și `false` ale tipului primitiv `boolean`

Computer Science



# Constante și metode statice în clasele învelitoare

- Metode statice pentru conversia unei reprezentări corect formate ca sir de caractere a unui număr la numărul de un tip dat
  - Metodele `Integer.parseInt`, `Long.parseLong`, `Float.parseFloat` și `Double.parseDouble` sunt pentru (în ordine) `int`, `long`, `float` și `double`
  - Exemplu

```
double x = Double.parseDouble("123.9");
```
- Metode statice pentru conversia duală celei anterioare
  - Exemplu

```
String x = Double.toString(123.99);
```
- Clasa `Character` conține un număr de metode utile la prelucrarea sirurilor de caractere



# Referințe Java

- Toate obiectele sunt create în memorie (pe *heap* – zona de memorie unde se face alocarea dinamică)
- Pentru a accesa datele dintr-un obiect sau a lucra cu un obiect, este nevoie de o variabilă pe stivă, variabilă care poate stoca o referință la adresa obiectului
- Despre variabilele care stochează referințe la adrese de obiecte se spune că păstrează tipul de date *referință*
- Exemplu

```
PlatitorTaxe t = new PlatitorTaxe(1111111120633,  
                                30000);
```

Computer Science



# Clasa **String**

- Nu există un tip primitiv pentru şiruri în Java
- Clasa **String** este o clasă Java predefinită folosită la stocarea și prelucrarea şirurilor de caractere
- Obiectele de tipul **String** sunt compuse din şiruri de caractere scrise între ghilimele
  - Orice şir scris între ghilimele este de clasă **String**  
**"Curs de POO"**
- Unei variabile de tipul **String** î se poate da valoarea unui obiect **String**  
**String s = "Curs de POO";**



# Concatenarea sirurilor de caractere

- *Concatenare*: folosind operatorul `+` operator aplicat asupra a două siruri pentru a le conecta și a forma un sir mai lung
  - Dacă

```
String salut = "Bună ziua, ", și
String javaClass = "viitori colegi",
atunci salut + javaClass
este stringul "Bună ziua, viitori colegi"
```
- Orice număr de siruri pot fi concatenate
- La combinarea unui sir de caractere cu aproape orice alt tip, rezultatul este un sir de caractere
  - `"Răspunsul este " + 42` se evaluatează la  
`"Răspunsul este 42"`



# Metode ale clasei String

- Clasa **String** conține multe metode utile la aplicațiile care prelucrează siruri de caractere
  - O metodă a clasei **String** se apelează prin numele unui obiect **String**, un punct, numele metodei și o paranteză întră care sunt cuprinse argumentele (dacă sunt)
  - O valoare returnată de o metodă **String** se poate folosi oriunde se poate folosi o valoare de tipul returnat
- ```
String greeting = "Hello";
int count = greeting.length();
System.out.println("Length is " +
    greeting.length());
```
- *Pozitia sau indexul* unui caracter într-un sir se numără întotdeauna de la zero



# Metode ale clasei `String`

## Lungimea sirului

|     |                         |                     |
|-----|-------------------------|---------------------|
| i = | <code>s.length()</code> | lungimea sirului s. |
|-----|-------------------------|---------------------|

## Comparații (notă: folosiți aceste metode în loc de == și !=)

|     |                                       |                                                                 |
|-----|---------------------------------------|-----------------------------------------------------------------|
| i = | <code>s.compareTo(t)</code>           | compară cu t. Returnează <0 dacă s<t,<br>0 dacă ==, >0 dacă s>t |
| i = | <code>s.compareToIgnoreCase(t)</code> | la fel ca mai sus, dar fără a ține seama de majuscule/minuscule |
| b = | <code>s.equals(t)</code>              | true dacă cele două siruri au valori egale                      |
| b = | <code>s.equalsIgnoreCase(t)</code>    | la fel ca mai sus, dar fără a ține seama de majuscule/minuscule |
| b = | <code>s.startsWith(t)</code>          | true dacă s începe cu t                                         |
| b = | <code>s.startsWith(t, i)</code>       | true dacă t apare începând cu indexul i                         |
| b = | <code>s.endsWith(t)</code>            | true dacă s se termină cu t                                     |



# Metode ale clasei `String`

**Căutare (notă : Toate metodele "indexOf" returnează -1 dacă sirul/characterul nu este găsit)**

|     |                                  |                                                                    |
|-----|----------------------------------|--------------------------------------------------------------------|
| i = | <code>s.indexOf(t)</code>        | indexul primei apariții lui String t în s.                         |
| i = | <code>s.indexOf(t, i)</code>     | indexul lui String t la sau după poziția i în s.                   |
| i = | <code>s.indexOf(c)</code>        | indexul primei apariții caracterului c în s.                       |
| i = | <code>s.indexOf(c, i)</code>     | indexul caracterului c la sau după poziția i din s.                |
| i = | <code>s.lastIndexOf(c)</code>    | indexul ultimei apariții lui c în s.                               |
| i = | <code>s.lastIndexOf(c, i)</code> | indexul ultimei apariții lui c pe sau înainte de poziția i în s.   |
| i = | <code>s.lastIndexOf(t)</code>    | indexul ultimei apariții lui t în s.                               |
| i = | <code>s.lastIndexOf(t, i)</code> | indexul ultimei apariții a lui t pe sau înainte de poziția i în s. |



# Metode ale clasei **String**

## Obținerea de părți

|      |                                |                                                                                 |
|------|--------------------------------|---------------------------------------------------------------------------------|
| c =  | <code>s.charAt(i)</code>       | caracterul de la poziția <i>i</i> din <i>s</i> .                                |
| s1 = | <code>s.substring(i)</code>    | subșirul de la indexul <i>i</i> până la sfârșitul lui <i>s</i> .                |
| s1 = | <code>s.substring(i, j)</code> | subșirul de la indexul <i>i</i> până înainte de indexul <i>j</i> din <i>s</i> . |

## Crearea unui nou sir din original

|      |                                |                                                               |
|------|--------------------------------|---------------------------------------------------------------|
| s1 = | <code>s.toLowerCase()</code>   | nou String cu toate caracterele minuscule                     |
| s1 = | <code>s.toUpperCase()</code>   | nou String cu toate caracterele majuscule                     |
| s1 = | <code>s.trim()</code>          | nou String fără spații albe la început și sfârșit             |
| s1 = | <code>s.replace(c1, c2)</code> | nou String cu toate <i>c2</i> -urile înlocuite prin <i>c1</i> |



# Metode ale clasei **String**

## Metode statice pentru conversia la String

|     |                                     |                                                                                                |
|-----|-------------------------------------|------------------------------------------------------------------------------------------------|
| s = | <code>String.valueOf(x)</code>      | Convertește x la String, unde x este orice valoare de tip (primitiv sau obiect).               |
| s = | <code>String.format(f, x...)</code> | [Java >=5] Folosește formatul f pentru a converti un număr variabil de parametri, x la un sir. |

- Lista nu este exhaustivă



# Seturi de caractere

- *ASCII*: Set de caractere folosit de multe limbaje de programare, care conține toate caracterele folosite în mod normal pe o tastatură pentru limba engleză plus câteva caractere speciale
  - Fiecare caracter este reprezentat de un cod numeric
- *Unicode*: Set de caractere folosit de limbajul Java care include tot setul ASCII plus multe dintre caracterele folosite cu alfabetele nelatine
  - Exemple

```
char c='\u0103'; // litera 'ă'  
String s="\u00eencoto\u015fm\u0103ni\u0163i"; // încotoșmăniți
```



# O porțiune din codul Unicode

|   | 000         | 001         | 002        | 003       | 004       | 005       | 006       | 007       |
|---|-------------|-------------|------------|-----------|-----------|-----------|-----------|-----------|
| 0 | NUL<br>0000 | DLE<br>0010 | SP<br>0020 | 0<br>0030 | @<br>0040 | P<br>0050 | `<br>0060 | p<br>0070 |
| 1 | SOH<br>0001 | DC1<br>0011 | !<br>0021  | 1<br>0031 | A<br>0041 | Q<br>0051 | a<br>0061 | q<br>0071 |
| 2 | STX<br>0002 | DC2<br>0012 | "<br>0022  | 2<br>0032 | B<br>0042 | R<br>0052 | b<br>0062 | r<br>0072 |
| 3 | ETX<br>0003 | DC3<br>0013 | #<br>0023  | 3<br>0033 | C<br>0043 | S<br>0053 | c<br>0063 | s<br>0073 |
| 4 | EOT<br>0004 | DC4<br>0014 | \$<br>0024 | 4<br>0034 | D<br>0044 | T<br>0054 | d<br>0064 | t<br>0074 |



# Numirea constantelor

- În loc de numere "anonyme", declarați constante simbolice (cu nume) și folosiți-le numele

```
public static final double CM_PER_INCH = 2.54;  
public static final int HOURS_PER_DAY = 24;
```

- Previne schimbarea nedorită a valorii
- Ușurează modificarea valorii

- Convenția de nume pentru constante

- Toate literele majuscule
- Limitele de cuvinte marcate prin liniuță de subliniere

Computer Science



# Comentariile

- Comentariu de o *linie*
  - Începe cu simbolurile `//` și provoacă ignorarea a ceea ce urmează până la sfârșitul liniei
  - Folosit de cel care scrie codul sau de cel care îl modifică
- Comentariul *bloc*
  - La fel ca în C (perechea `/*, */`)
  - Furnizează documentație utilizatorilor programului



# Documentarea programului

- Java include programul **javadoc** care extrage automat documentația din comentariile bloc din clasele definite, dacă
  - Începutul comentariului are un asterisc suplimentar (**/\*\***)
- Un program bine scris este autodocumentat
  - Structura sa se clarifică prin alegerea numelor de identificatori și modelul de indentare



# Operatori

- Sunt tratați în detaliu în lucrarea de laborator
- Câteva diferențe față de C:
  - Concatenarea pentru String: operatorul +
  - Operatorul pe biți >>>
    - D.e. `n >>> p;` // deplasează biții lui *n* spre dreapta cu *p* poziții.  
În pozițiile de rang superior se inserează zerouri.
  - Operatori pentru lucrul cu obiecte – vor fi tratați în detaliu mai târziu

Computer Science



# Despre precedența operatorilor

## Precedența operatorilor

|        |            |               |                                                  |
|--------|------------|---------------|--------------------------------------------------|
| .      | [ ] (args) | post ++ --    | Tineți minte precedența pentru operatorii unari, |
| !      | ~ unary    | + - pre ++ -- | * / %                                            |
| (type) | new        |               | + -                                              |
| *      | / %        |               | <i>comparații</i>                                |
| +      | -          |               | &&                                               |
| <<     | >>         | >>>           | = <i>asignări</i>                                |
| <      | <=         | >             | Folosiți paranteze () pentru ceilalți            |
| >=     | instanceof |               |                                                  |
| ==     | !=         |               |                                                  |
| &      |            |               |                                                  |
| ^      |            |               |                                                  |
|        |            |               |                                                  |
| &&     |            |               |                                                  |
|        |            |               |                                                  |
| ? :    |            |               |                                                  |
| =      | += -=      | etc           |                                                  |



# Instrucțiunea **if**

- Instrucțiunea **if** specifică ce bloc de cod să se execute în funcție de rezultatul evaluării unei condiții numită *expresie booleană*

```
if (<expresie booleană>)
    <then> Instr1
else
    <else> Instr 2
```

Sau:

```
(<expresie booleană>) ? Instr1 : Instr2
ex: int max = (a>b) ? a:b;
```

- **<expresie booleană>** este o expresie conditională care se valuează la true sau false.
  - Sintaxă similară limbajului C, dar atenție la ce este o expresie booleană în Java



## Compararea obiectelor

- La compararea a două **variabile**, se compară *conținutul* lor
- În cazul **obiectelor**, *conținutul* este **adresa** unde este stocat obiectul (adică se vor compara referințele)
  - Sirurile de caractere în Java sunt obiecte ale clasei String
  - Clasa String furnizează metode de comparare
- Cea mai bună metodă în ceea ce privește compararea obiectelor este să *definim metode de comparare* pentru clasa respectivă



# Sugestii pentru **if**

- Începeți testul cu cazul cel mai relevant
  - Face codul mai ușor de citit
- Nu uitați de clauza else!
- Evitați condițiile complicate
- Divizați condițiile în variabile/funcții booleene
- Încercați să folosiți condiții pozitive
- Exemplu – se preferă a doua variantă

```
if (!node.isFirst() && node.value() != null)
    instr1
else
    instr2

if (node.isFirst() || node.value() == null)
    instr2
else
    instr1
```



# Instrucțiunea **switch**

## ■ Sintaxa pentru instrucțiunea **switch**

```
switch ( < expresie aritmetică> ) {  
    <case eticheta_1>: <case corp 1>  
    ...  
    <case eticheta_n>: <case corp n>  
}
```

- Tipul de dată al **<expresie aritmetică>** trebuie să fie **char, byte, short, int**

Computer Science



# Instructiunea switch

## ■ Exemplu

```
int month = 8;
String monthString;
switch (month) {
    case 1: monthString = "January";
              break;
    case 2: monthString = "February";
              break;
    //. . .
    case 12: monthString = "December";
              break;
    default: monthString = "Invalid month";
              break;
}
System.out.println(monthString);
```



# Instrucțiunea **switch**

- Dacă există o valoare care se potrivește cu valoarea expresiei, atunci se execută corpul acesteia. Altfel execuția continuă cu instrucțiunea care urmează instrucțiunii **switch**
- Instrucțiunea **break** face să nu se execute porțiunea care urmează din **switch**, ci să se continue cu ceea ce vine după **switch**
- **break** este necesar pentru a executa instrucțiunile dintr-un caz, și numai unul
  - iarăși, ca în C

Computer Science



# Sugestii pentru **switch**

- Ordonați cazurile (logic sau alfabetic)
  - Prevedeți întotdeauna cazul implicit (**default**)
  - Întotdeauna folosiți **break** între cazuri
  - Încercați să păstrați mică dimensiunea instrucțiunii **switch**
  - Divizați cazurile de mari dimensiuni în funcții



# Instrucțiuni repetitive

- *Instrucțiunile repetitive* controlează un bloc de cod de executat un număr fixat de ori sau până la îndeplinirea unei anumite condiții
- Ca și C, Java are trei instrucțiuni repetitive:
  - **while**
  - **do-while**
  - **for**
- Instrucțiunile repetitive sunt numite și *instrucțiuni de ciclare*, blocul de instrucțiuni care se repetă (*<instructiuni>* în cele ce urmează) este cunoscut sub numele de *corpușul ciclului*

Computer Science



# Instrucțiunea **while**

- În Java, instrucțiunile **while** au formatul general:

```
while ( <expresie booleana> )
      <instructiuni>
```
- Cât timp **<expresie booleana>** este true, se execută corpul ciclului
- Într-o *buclă controlată prin contor*, corpul ciclului este executat de un număr fixat de ori
- Într-o buclă *controlată printr-o sentinelă*, corpul ciclului este executat în mod repetat până când se întâlnește o valoare stabilită, numită *sentinelă*

Computer Science



# Capcane în scrierea instrucțiunilor repetitive

- La instrucțiunile repetitive este important să se asigure terminarea ciclului la un moment dat
- Tipurile de probleme potențiale de ținut minte:
- **Bucla infinită**

```
int item = 0;  
while (item < 5000) {  
    item = item * 5;  
}
```

- Deoarece **item** este inițializat la 0, **item** nu va fi niciodată mai mare de 5000 ( $0 = 0 * 5$ ), astfel că bucla nu se va termina niciodată



# Capcane în scrierea instrucțiunilor repetitive

## ■ Bucla infinită

```
int count = 1;  
while (count != 10)  
{  
    count = count + 2;  
}
```

- În acest exemplu, (al cărui instrucțiune **while** este în buclă infinită), contorul va fi 9 și 11, dar nu 10.
- **Eroarea prin depășire de capacitate** apare atunci când se încearcă asignarea unei valori mai mari decât valoarea maximă pe care o poate stoca variabila



# Capcane în scrierea instrucțiunilor repetitive

- **Depășirea de capacitate**
- În Java, o depășire a capacitatii de reprezentare nu provoacă terminarea programului
  - La tipurile **float** și **double** se atribuie variabilei o valoare care reprezintă infinit
  - La tipul **int**, o valoare pozitivă devine negativă, iar una negativă devine pozitivă, ca și cum valorile ar fi plasate pe un cerc cu maximul și minimul învecinate
- Numerele reale nu se folosesc în teste exacte sau incrementări, deoarece valorile lor sunt reprezentate cu aproximatie
- **Eroarea prin depășire cu unu** este o altă capcană frecventă



# Instrucțiunea **do-while**

- Instrucțiunea **while** este o *bucătă cu pre-testare* (testul se face la intrarea în buclă)
  - Corpul buclei poate să nu fie executat niciodată
- Instrucțiunea **do-while** este o *bucătă cu post-testare*
  - Corpul ciclului este executat cel puțin o dată
- Formatul instrucțiunii **do-while** este:

```
do
    <instructiuni>
  while (<expresie booleană>);
```
- **<instructiuni>** se execută până când **<expresie booleană>** devine falsă



# Controlul repetitiv o buclă-și-jumătate (buclă infinită întreruptă)

- Atenție la două lucruri la folosirea buclelor întrerupte
  - **Pericolul ciclării infinite.** Expresia booleană a instrucțiunii `while` este `true` întotdeauna. Dacă nu există o instrucțiune `if` care să forțeze ieșirea din ciclu, rezultatul va fi o buclă infinită
  - **Puncte de ieșire multiple.** Se poate totuși, chiar dacă e mai complicat să se scrie un control buclă cu mai multe ieșiri (`break`). Practica recomandă pe cât posibil curgerea *o-intrare o-iesire* a controlului



# Instrucțiunea **for**

- Formatul instrucțiunii **for** este:

```
for (<initializare>; <expresie booleană>; <increment>)
    <instrucțiuni>
```

- Exemplu:

```
int sum = 0;
for (int i = 1; i <=100; i++) {
    sum += i;
}
```

- Variabila **i** din exemplu se numește *variabilă de control* (contorizează numărul de repetiții)
- **<increment>** poate fi orice cantitate
- Din nou, ca în C



## Sugestii pentru ciclurile **for**

- Sunt ideale atunci când numărul de iterații este cunoscut
  - Folosiți o instrucțiune pentru fiecare parte
  - Declarați variabilele de ciclu în antet (reduce vizibilitatea și interferența)
  - Nu se recomandă modificarea variabilei de control în corpul ciclului

Computer Science



# break cu etichetă

- **break** se folosește în **bucle** și **switch**
  - semnificațiile sunt diferite
- **break** poate fi și urmat de o etichetă, *L*
  - încearcă să transfere controlul la o instrucțiune etichetată cu *L*
  - Dacă nu există instrucțiuni etichetate cu *L*, apare o eroare de compilare
  - Un **break** cu etichetă permite ieșirea din mai multe bucle imbricate
  - Eticheta trebuie să *precedă* bucla cea mai exterioară din care se dorește ieșirea
  - Această formă nu există în C



# Exemplu pentru **break** cu etichetă

```
int n;
read_data:
while(...) {
    ...
    for (...) {
        n= Console.readInt(...);
        if (n < 0) // nu se poate continua
            break read_data; // break out of data loop
        ...
    }
}
// verifica dacă este succes sau eșec
if (n < 0) {
    // tratează situațiile cu probleme
}
else {
    // am ajuns aici normal
}
```



## continue cu etichetă

- Forma etichetată a instrucțiunii **continue** sărăcășește peste iterația curentă a unei bucle exterioare marcate cu o etichetă dată
- Eticheta trebuie să *preceadă* bucla cea mai exterioară din care se dorește ieșirea

Computer Science

```
public class ContinueWithLabelDemo {  
    public static void main(String[] args) {  
        String searchMe = "Look for a substring in me";  
        String substring = "sub";  
        boolean foundIt = false;  
        int max = searchMe.length() - substring.length();  
        test:  
            for (int i = 0; i <= max; i++) {  
                int n = substring.length();  
                int j = i;  
                int k = 0;  
                while (n-- != 0) {  
                    if (searchMe.charAt(j++) != substring.charAt(k++)) {  
                        continue test;  
                    }  
                }  
                foundIt = true;  
                break test;  
            }  
        System.out.println(foundIt ? "Found it" : "Didn't  
                                    find it");  
    }  
}
```



# for pentru iterații peste colecții și tablouri

- Creat special pentru iterații peste colecții și tablouri (vom reveni asupra instrucțiunii) – Java 5
- Nu funcționează oriunde (d.e. nu se pot accesa indicii de tablouri)
- Exemplu

```
public class ForEachDemo {  
    public static void main(String[] args) {  
        int[] arrayOfInts = {32, 87, 3, 589, 12, 1076};  
        for (int element : arrayOfInts) {  
            System.out.print(element + " ");  
        }  
        System.out.println();  
    }  
}
```

Afișare la ieșirea standard:

32 87 3 589 12 1076



# Anatomia unei clase

TECHNICAL UNIVERSITY

```
public class Taxi{  
    private int km;  
    public Taxi(){  
        km = 0;  
    }  
  
    public int getKm(){  
        return km;  
    }  
    public void drive(int km){  
        this.km += km;  
    }  
}
```

*Antetul clasei*

*Variabile instanță (câmpuri)*

*Constructori*

*Metode*



# Constructor:

TECHNICAL UNIVERSITY

|              |                                                                                                                                                                                                                                                                                |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Scop         | Initializează starea unui obiect: <ul style="list-style-type: none"><li>- se initializează toate atributele obiectului</li><li>- dacă acestea nu se initializează explicit, ele vor fi initializează implicit cu valorile inițiale din Java, în funcție de tipul lor</li></ul> |
| Nume         | La fel cu numele clasei<br>Prima literă mare                                                                                                                                                                                                                                   |
| Cod          | <code>public Taxi () {...}</code>                                                                                                                                                                                                                                              |
| Ieșire       | Nu este tip de return în antet                                                                                                                                                                                                                                                 |
| Intrare      | 0 sau mai mulți parametri                                                                                                                                                                                                                                                      |
| Utilizare    | > <code>Taxi cab;</code><br>> <code>cab = new Taxi();</code>                                                                                                                                                                                                                   |
| # de apelări | Cel mult o dată pe obiect; invocat de operatorul "new"                                                                                                                                                                                                                         |



# Constructori multipli

TECHNICAL UNIVERSITY

```
public class Taxi{  
    private int km;  
    private String driver;  
  
    public Taxi(){  
        km = 0;  
        driver = "Unknown";  
    }  
  
    public Taxi(int km, String d){  
        this.km = km;  
        driver = d;  
    }  
}
```

O operație "new" încununată de succes creează un obiect pe heap și execută constructorul al cărui listă de parametri "corespunde" listei sale de argumente (ca număr, tip, ordine).

```
> Taxi cab1;  
> cab1 = new Taxi();  
  
> Taxi cab2;  
> cab2 = new Taxi(10, "Jim");
```



# Folosirea corespunzătoare a constructorilor

- Un constructor ar trebui *întotdeauna* să creeze obiectele într-o stare *validă*
  - Constructorii nu trebuie să facă nimic altceva decât să creeze obiecte
  - Dacă un constructor nu poate garanta că obiectul construit este valid, atunci ar trebui să fie **private** și accesat prin intermediul unei metode de fabricare
  - Notă: În general, termenul de **metodă de fabricare** este folosit pentru a se referi la orice metodă al cărei scop principal este crearea de obiecte



# Folosirea corespunzătoare a constructorilor

- O **metodă de fabricare (factory method)**: metodă statică care apelează un constructor

- Constructorul este de obicei **private**
- Metoda de fabricare poate determina dacă să învoce sau nu constructorul
- Metoda de fabricare poate arunca o **excepție**, sau poate face altceva potrivit în cazul în care i se dau argumente ilegale sau nu poate crea un obiect valid
- Exemplu

```
public static Person create(int age) {  
    if (age < 0)  
        throw new IllegalArgumentException("Too young!");  
    else  
        return new Person(age);  
}
```



# Metodă:

|              |                                                         |
|--------------|---------------------------------------------------------|
| Scop         | Execută comportamentul obiectului                       |
| Nume         | Un <b>verb</b> ;<br>Începe cu literă mică               |
| Cod          | <code>public void turnLeft() {<br/>    ...<br/>}</code> |
| Ieșire       | Pentru ieșire este nevoie de un tip returnat            |
| Intrare      | 0 sau mai mulți parametri                               |
| Utilizare    | <code>&gt; cab.turnLeft();</code>                       |
| # de apelări | nelimitat pentru un obiect                              |



# Ce poate face o metodă?

- O metodă poate

- Schimba starea obiectului său
- Raporta starea obiectului său
- Opera asupra numerelor, a textului, a fișierelor, graficii, paginilor web, hardware, ...
- Crea alte obiecte
- Apela o metodă a unui alt obiect:  
*obiect.metoda(args)*
- Apela o metodă din aceeași clasă:  
*this.metoda(args);*
- Se poate autoapela (recursivitate)



# Declararea unei metode

```
public tip_returnat numeMetoda (0+ parametri) {  
    ...  
}
```

- **Nume:** Verb care începe cu literă mică
- **Ieșire:** e nevoie de un tip returnat
- **Intrare:** 0 sau mai mulți parametri
- **Corp:**
  - Între accolade
  - Conține un număr arbitrar de instrucțiuni
  - Poate conține declarații de “variabile locale”
- **Cum se apelează:** operatorul “punct”:  
*numeObiect.numeMetoda(argumente)*



# Metode accesoare și mutatoare

```
public class Taxi{  
    private int km;  
  
    public Taxi(){  
        km = 0;  
    }  
  
    // raportează # km  
    public int getKm(){  
        return km;  
    }  
  
    // setează (schimbă) # km  
    public void setKm(int m){  
        km = m;  
    }  
}
```

*Apeluri de metode  
accesoare (de obținere)/  
mutatoare(de setare)*

```
> Taxi cab1;  
> cab1 = new Taxi();  
> cab1.getKm()  
0  
> cab1.setKm(500);  
> cab1.getKm()  
500
```



# Intrarea pentru o metodă

- O metodă poate primi 0 sau mai multe intrări
- Intrările pe care le așteaptă o metodă sunt specificate via lista de "parametri formali" (`tip nume1, tip nume2, ...`)
- La apelul unei metode, numărul, ordinea și tipul argumentelor trebuie să se potrivească cu parametrii corespunzători

| Declararea metodei<br>(cu parametri)                        | Apelul metodei<br>(cu argumente)  |
|-------------------------------------------------------------|-----------------------------------|
| <code>public void meth1() { ... }</code>                    | <code>obj.meth1()</code>          |
| <code>public int meth2(boolean b) { ... }</code>            | <code>obj.meth2(true)</code>      |
| <code>public int meth3(int x, int y, Taxi t) { ... }</code> | <code>obj.meth3(3, 4, cab)</code> |



# Ieșirea unei metode

- O metodă poate să nu aibă ieșire (void) sau să aibă ceva
- Dacă nu returnează nimic (nu are ieșire), tipul returnat este "void"

```
public void setKm(int km) { . . }
```

- Dacă are ieșire:
    - Tipul returnat este non-void (d.e. int, boolean, Taxi)
- ```
public int getKm() { . . }
```
- Trebuie să conțină o instrucțiune return cu o valoare
    - // valoarea returnată trebuie să se
    - // potrivească cu tipul returnat
    - return km;**



# Modificatori de acces pentru metode

- **private** – nu poate fi folosită de toate clasele; metoda este vizibilă doar în cadrul clasei
- **default (nu se specifică nimic)** – nu poate fi folosită de toate clasele; metoda este vizibilă doar în pachetul din care face parte clasa
- **protected** – nu poate fi folosită de toate clasele; metoda este vizibilă doar în pachetul din care face parte clasa și în subclasele acesteia chiar dacă sunt în alte pachete
- **public** – cel mai frecvent folosit; metoda este vizibilă tuturor
- **static** – nu e nevoie de obiecte pentru a folosi acest fel de metode
  - Dacă declarația metodei conține modificadorul **static**, este vorba de o metodă la nivelul clasei
  - Metodele la nivelul clasei pot accesa doar constantele și variabilele clasei



# Suprîncărcarea unei metode

## Suprîncărcare:

- Implică folosirea unui termen pentru a indica semnificații diverse
- Scrierea unei metode cu același nume, dar cu *argumente diferite*
- Suprîncărcarea unei metode Java înseamnă scrierea mai multor metode cu același nume
- Exemplu:

```
public int test(int i, int j){  
    return i + j; }  
public int test(int i, byte j){  
    return i + j; }
```



# Ambiguitate la supraîncărcare

- La supraîncărcarea unei metode există riscul **ambiguității**
- O situație ambiguă este când compilatorul nu poate determina ce metodă să folosească
- Exemplu:

```
public int test(int units, int pricePerUnit)
{
    return units * pricePerAmount;
}

public long test(int numUnits, int weight)
{
    return (long) (units * weight);
}
```

- O supraîncărcare validă necesită cel puțin un argument de tip diferit sau un număr diferit de argumente



# Supraîncărcarea constructorilor

- Java furnizează automat un constructor la crearea unei clase
- Programatorii pot scrie constructori proprii, inclusiv constructori care primesc argumente
  - Asemenea argumente sunt folosite la inițializare atunci când valorile obiectelor pot diferi
- Exemplu

```
class Client {  
    private String nume;  
    private int numarCont;  
    public Client(String n) {  
        nume = n;  
    }  
    public Client(String n, int a) {  
        nume = n;  
        numarCont = a;  
    }  
}
```



# Supraîncărcarea constructorilor

- Dacă o clasă este instanțială, Java furnizează automat un constructor
- La crearea unui constructor propriu, constructorul furnizat automat *încetează să existe*
- Ca și la alte metode, constructorii pot fi *supraîncărcați*
  - Supraîncărcarea constructorilor oferă o cale de a crea obiecte cu sau fără argumente inițiale, după necesități

Computer Science



# Referință **this**

- Compilatorul accesează câmpurile de date corespunzătoare ale obiectului, deoarece i se trimit implicit o referință **this** la câmpurile și metodele claselor

```
class Student{  
    private int studentID;  
    public Student(int studentID){  
        this.studentID = studentID;  
    }  
    public Student(){  
        this(0);  
    }  
    public getStudentID(){  
        return this.studentID; //return studentID;  
    }  
}
```

- Metodele statice (metodele la nivel de clasă) nu au o referință **this** deoarece nu au un obiect asociat



# Clasă instantiabilă

- O clasă este *instantiabilă* dacă putem crea instanțe ale clasei respective
- Exemple: clasele învelitoare ale tipurilor primitive, **String**, **Scanner**,... sunt toate instantiabile, în timp ce clasa **Math** nu este
- Fiecare obiect este membru al unei clase
  - catedra este un obiect membru al clasei Catedra
  - relații de tipul **este-o**



# Utilitatea conceptului de clasă

- Clasa reprezintă şablonul cu atributele şi funcionalităţile obiectelor
- Toate obiectele au atribute predictibile deoarece obiectele sunt membre ale anumitor clase
- Pentru crearea unui obiect de tipul unei clase, trebuie să creaţi clase din care să se instantieze obiectele
  - Exemplu - clasa Taxi şi instanţă t: Taxi t = new Taxi();
- Se pot scrie alte clase care să folosească obiectele
  - Exemplu - se scrie un program/o clasă pentru a conduce taxiul la aeroport; foloseşte clasa Taxi pentru a crea un obiect Automobil de condus



# Crearea unei clase

- Trebuie:
  - Să dați un nume clasei
  - Să determinați ce date și metode vor fi parte a clasei
- Modificatorii de acces pentru clase cuprind:
  - **public**
    - Cel mai folosit; clasa este vizibilă tuturor
    - Clasa poate fi **extinsă** sau folosită ca bază pentru alte clase
  - **final** – folosit doar în circumstanțe speciale (clasa este complet definită și nu sunt necesare definiri de subclase)
  - **abstract** – folosit doar în circumstanțe speciale (clasa este incomplet definită – conține o metodă neimplementată)



# Şablon de program pentru codul unei clase

TECHNICAL UNIVERSITY

## Clauze import

## Comentariu pentru clasă

Descriere a clasei în formatul pentru javadoc

```
class  {
```

## Numele clasei

## Declarații

Datele partajate de mai multe metode se declară aici

## Metodă

...

## Metodă

```
}
```



# Blocuri și vizibilitate (scop)

- **Bloc**: În orice clasă sau metodă, conține codul inclus între o pereche de acolade
- Portiunea de program în care se poate referi o variabilă constituie **domeniul de vizibilitate (scop)** al variabilei
- O variabilă există, sau devine vizibilă la declarare
- O variabilă începează să existe sau devine invizibilă la sfârșitul blocului în care a fost declarată
- Dacă declarați o variabilă într-o clasă și folosiți același nume pentru a declara o variabilă într-o metodă a clasei, variabila declarată în metodă are precedență sau **suprascrie**, prima variabilă



# Variabile la nivel de clasă

- Variabile la nivel de clasă: variabile care sunt partajate de fiecare instanță a unei clase
- Numele instituției = "T.U. Cluj-Napoca"
- Fiecare angajat al unei instituții lucrează pentru aceeași instituție

```
private static String COMPANY_ID =  
    "T.U. Cluj-Napoca";
```

- Se poate dar nu este recomandat să se declare o variabilă vizibilă în afara clasei

Computer Science



# Folosirea constantelor și metodelor importate automat, de bibliotecă

- Creatorii limbajului Java au creat cca. 500 clase
  - Exemple: System, Scanner, Character, Boolean, Byte, Short, Integer, Long, Float și Double sunt clase
- Aceste clase sunt stocate în pachete (biblioteci) de clase
- Un pachet este un instrument de grupare convenabilă a claselor cu funcționalitate înrudită
- **java.lang** – pachetul este importat implicit în fiecare program Java care conține clase fundamentale (de bază) – d.e. System, Character, Boolean, Byte, Short, Integer, Long, Float, Double, String



# Folosirea metodelor importate, de bibliotecă

- Pentru a folosi oricare dintre clasele de bibliotecă (altele decât `java.lang`):
    - Folosiți întreaga cale împreună cu numele clasei  
`area = Math.PI * radius * radius;`  
sau
    - Importați clasa
  
sau
  - Importați pachetul care conține clasa pe care o folosiți
- Pentru a importa un întreg pachet folosiți simbolul de nume global, \*
- Exemplu
  - `import java.util.*;`
  - Reprezintă toate clasele dintr-un pachet



# Metode statice

- În Java se pot declara metode și variabile care să aparțină *clasei*, nu obiectului. Aceasta se face prin declararea lor ca **static**
- Metodele statice: declarate inserând cuvântul cheie **static** imediat după specificatorul de vizibilitate

```
public class ArrayWorks {  
    public static double medie(int[] arr) {  
        double total = 0.0;  
        for (int k=0; k<arr.length; k++) {  
            total = total + arr[k];  
        }  
        return total / arr.length;  
    }  
}
```



# Metode statice

- Metodele statice sunt apelate folosind numele clasei în locul unei referințe la un obiect

```
int[] myArray = {10, 21, 1};  
...  
double media = ArrayWorks.media(myArray);
```

- Exemple metode statice utile: în clasa Java standard, numită **Math**

```
public class Math {  
    public static double abs(double d) {...}  
    public static int abs(int k) {...}  
    public static double pow(double b, double exp) {...}  
    public static double random() {...}  
    public static int round(float f) {...}  
    public static long round(double d) {...}  
    ...  
}
```



# Restrictii pentru metodele statice

- Corpul unei metode statice nu poate referi nici o variabilă instanță (ne-statică)
- Corpul unei metode statice nu poate invoca nici o metodă ne-statică
- Dar, corpul unei metode statice poate instanția obiecte

```
public class Go {  
    public static void main(String[] args) {  
        Greeting greeting = new Greeting();  
    }  
}
```

- Aplicațiile Java de sine stătătoare, trebuie să-și inițieze execuția dintr-o metodă statică numită întotdeauna *main* și care are un singur tablou de String-uri ca parametru



# Variabile statice

- Orice *variabilă instanță* poate fi declarată **static** prin includerea cuvântului-cheie **static** imediat înainte de specificarea tipului său

```
public class StaticStuff {  
    public static double staticDouble;  
    public static String staticString;  
    . . .  
}
```

- O variabilă statică
  - Poate fi referită fie de clasă fie de obiect
  - Instantierea unui nou obiect de același tip nu sporește numărul de variabile statice



# Exemplu cu variabile statice

```
StaticStuff s1, s2;  
s1 = new StaticStuff();  
s2 = new StaticStuff();  
s1.staticDouble = 3.7;  
System.out.println( s1.staticDouble );  
System.out.println( s2.staticDouble );  
s1.staticString = "abc";  
s2.staticString = "xyz";  
System.out.println( s1.staticString );  
System.out.println( s2.staticString );
```



# Exemple cu constante

- Exemplu de constante în clasa standard Java numită **Color**

```
public class Color {  
    public static final Color BLACK = new Color(0,0,0);  
    public static final Color BLUE = new Color(0,0,255);  
    public static final Color GREEN= new Color(0,255,0);  
    . . .  
}
```

- Constantele Color

- Ajută la compararea culorilor folosind numele asignat lor

```
Color myColor;
```

```
....
```

```
if (myColor == Color.GREEN) ...
```



# Programare orientată pe obiecte

1. Clase și obiecte (continuare)
2. Tablouri



# Metode: cum funcționează un apel

```
// Autor : Fred Swartz
import javax.swing.*;
public class KmToMiles {
    private static double convertKmToMi(double kilometri) {
        return kilometri * 0.621; // sunt 0.621 mile intr-un kilometru.
    }
    public static void main(String[] args) {
        //... variabile locale
        String kmStr;      // String km înainte de conversia la double.
        double km;          // Număr of kilometri.
        double mi;          // Număr of mile.
        //... Intrare
        kmStr = JOptionPane.showInputDialog(null, "Introduceti kilometri.");
        km = Double.parseDouble(kmStr);
        //... Calcule
        mi = convertKmToMi(km);
        //... Output
        JOptionPane.showMessageDialog(null, km + " kilometri sunt " + mi + " mile.");
    }
}
```



# Metode: cum funcționează un apel

- Considerați atribuirea `mi=convertKmToMi(km)` ;
- Pașii pentru procesarea acestei instrucțiuni sunt:
  1. Evaluează argumentele de-la-stânga-la-dreapta
  2. Depune un nou cadru de stivă (stack frame) pe stiva de apeluri. Spațiu pentru parametri și variabilele locale (parametrul kilometri doar, aici). Starea salvată a metodei apelante (include adresa de return)
  3. Inițializează parametrii. La evaluarea argumentelor, acestea sunt asignate parametrilor locali din metoda apelată.
  4. Execută metoda.
  5. Revine din metodă. Memoria folosită pentru cadrul de stivă pentru metoda apelată este scoasă de pe stivă.



# Transmiterea parametrilor

- In Java transmiterea parametrilor la apelul metodelor se face **numai prin valoare**
  - Modificările aduse parametrilor în interiorul unei metode nu se păstrează la revenirea din metoda respectivă
- Dacă unul dintre parametrii unei metode are drept tip o clasă, aceasta înseamnă că la apel metoda va primi referința unui obiect al clasei
  - Ceea ce se transmite prin valoare este chiar referința, NU obiectul indicat de ea
  - Metoda **nu va putea modifica referința** respectivă, dar **va putea modifica obiectul** indicat de ea



# Crearea obiectelor

- Java are trei mecanisme dedicate asigurării inițializării corespunzătoare a obiectelor:
  - *inițializatori de instanță* (numiți și blocuri de inițializare de instanță)
  - *inițializatori de variabile instanță*
  - *constructori*
- Toate cele trei mecanisme presupun cod executat automat la crearea unui obiect
- La alocarea memoriei pentru un nou obiect folosind operatorul **new** sau metoda **newInstance()** a clasei **Class**, JVM asigură executarea codului de inițializare înainte de folosirea zonei alocate



# Crearea obiectelor

- La invocarea operatorului **new**
  - Se alocă memorie (se rezervă spațiu pentru obiect). Variabilele instanță sunt inițializate la valorile lor implicate
  - Se execută inițializarea explicită. Variabilele inițializate la declararea atributelor primesc valorile declarate
  - Se execută un constructor. Valorile variabilelor pot fi schimbate de constructor
  - Se atribuie variabilei o referință la obiect

## ■ Exemplu:

```
class Ex{  
  
    int nr = 1;  
  
    public Ex() {  
        nr = 20;  
    }  
  
    public static void main() {  
        Ex e = new Ex();  
        System.out.println(  
            "Nr = "+ e.nr);  
    }  
}
```



# Valori initiale implicate pentru câmpuri (variable instanță)

Tip	Valoare
boolean	false
byte	(byte) 0
short	(short) 0
int	0
long	0L
char	\u0000
float	0.0f
double	0.0d
referință la obiect	null



# Inițializarea câmpurilor

- Atribuire simplă
  - O valoare inițială unui câmp la declararea sa
  - Exemplu
    - `public static int capacity = 10; //initializat la 10`
    - `private boolean full = false; //initializat la false`
- Blocuri de inițializare statice
  - Bloc normal de cod între acolade, { } și precedat de cuvântul cheie static
  - Exemplu
    - `static {`  
          `// codul necesar inițializării se scrie aici`  
        `}`
    - Folosit la inițializarea variabilelor la nivel de clasă
    - Blocurile de inițializare statice sunt executate în ordinea în care apar în codul sursă



# Inițializarea câmpurilor

- Alternativă la blocurile statice: metodă statică privată
  - Exemplu

```
class Oricare {  
    public static varType myVar = initializeClassVariable();  
    private static varType initializeClassVariable() {  
        //codul de inițializare se pune aici  
    }  
}
```

- Inițializarea membrilor instanțelor
  - Asemănătoare blocurile statice, dar nu static. Compilatorul Java copiază blocurile de inițializare în fiecare constructor
  - Exemplu

```
{  
    // codul pentru inițializare, aici  
}
```



# Inițializarea câmpurilor

- Alternativă la inițializarea membrilor instanțelor
  - Se folosește o metodă **final** pentru inițializarea unei variabile instanță:

```
class Oricare {  
    private varType myVar = initializeInstanceVariable();  
    protected final varType initializeInstanceVariable(){  
        //cod de inițializare  
    }  
}
```

- Folositoare mai ales dacă subclasele doresc să refolosească metoda de inițializare
- Metoda este **final** deoarece apelul metodelor non-final la inițializarea instanțelor pot cauza probleme



# Crearea obiectelor

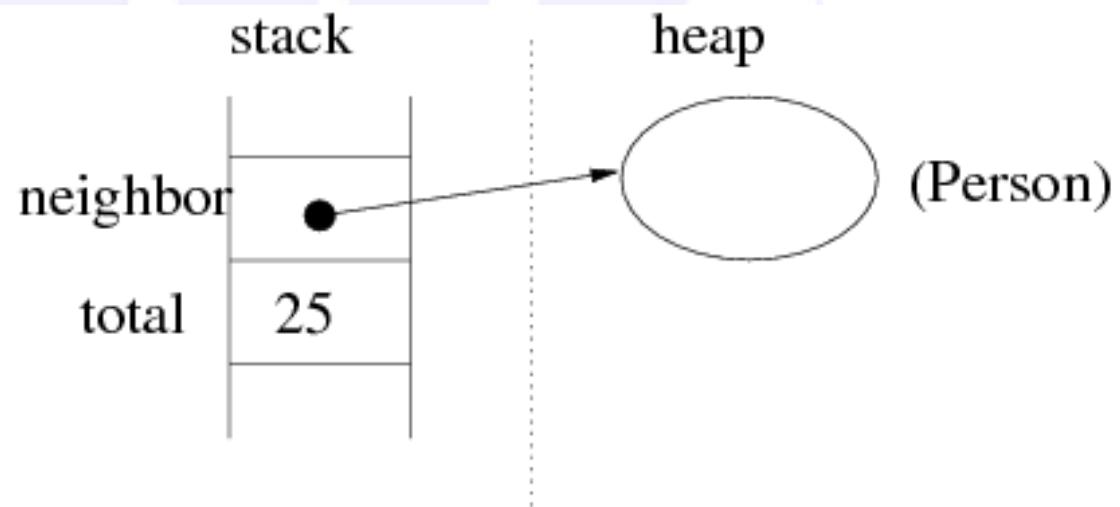
- La apelul unui constructor
  - Se alocă spațiu *pe heap* pentru obiect
  - *Fiecare obiect primește spațiu propriu (propria copie a variabilelor instanță)*
  - Starea obiectului este inițializată potrivit codului (definit de programator) clasei
- Declararea unei variabile ca fiind de un tip de obiect produce o referință la obiect și nu obiectul în sine
  - Pentru a obține obiectul în sine folosiți **new** și un constructor pentru clasă

Computer Science



# Crearea obiectelor

- Se poate combina declararea și inițializarea  
`Person neighbor = new Person();`
- La fel ca și pentru tipurile primitive  
`int total = 25;`





# Constructori

- Orice clasă (inclusiv cele abstracte) trebuie să aibă cel puțin un constructor
  - Nu înseamnă că trebuie neapărat implementat unul
  - În lipsa codului explicit, Java generează implicit un constructor, caz în care variabilele instantă vor fi initializate cu valorile implicite
- Exemplu de cod pentru implementarea unui constructor

```
class Person{  
    String nume;  
    int varsta;  
  
    public Person(String n, int v){  
        nume = n;  
        varsta = v;  
    }
```

```
public Person() {  
}
```

```
public static void main() {  
    Person p = new Person("Ion", 28);  
    // alt cod...  
}
```

Computer Science



# Asignarea obiectelor și alias-uri

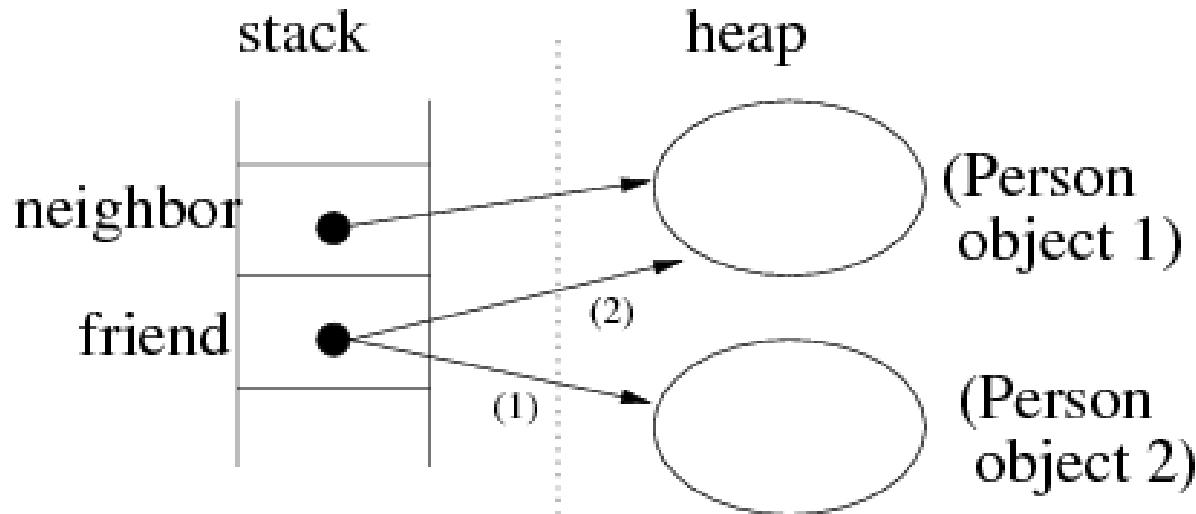
- Semnificația asignării este *diferită* pentru obiecte față de tipurile primitive

```
int num1 = 5;  
int num2 = 12;  
num2 = num1; // num2 conține acum 5  
//-----  
Person neighbor = new Person(); // creează object1  
Person friend = new Person(); // creează object2  
friend = neighbor;
```

- La sfârșit atât **friend** cât și **neighbor** se referă la object1 (ele sunt **alias** unul pentru celălalt) și nimic nu se mai referă la object2 (acesta este **inaccesibil**)



# Asignarea obiectelor și alias-uri



- Java va *colecta automat reziduul* (zona de memorie nefolosită) **object2**



# Alocare memorie pe **stack** și **heap**

- De obicei metodele, variabilele și obiectele din programele Java sunt alocate în una din cele două locuri de memorie: **stack** sau **heap**
- Regula de bază este:
  - Variabilele instantă și obiectele se alocă pe **heap**
  - Variabilele locale se alocă pe **stack**
- Exemplu concret de cum se alocă variabilele dintr-un program Java:

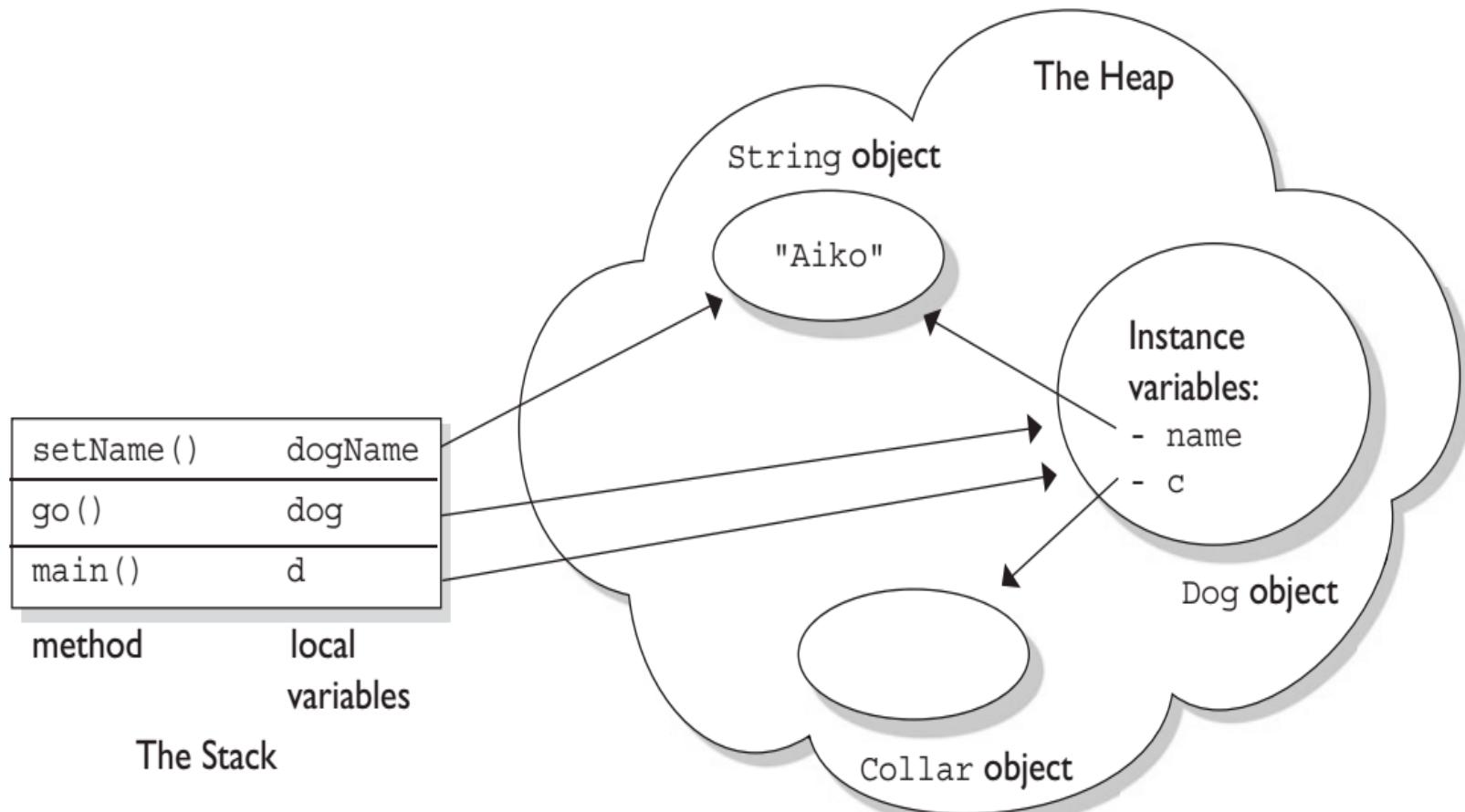


# Alocare memorie pe **stack** și **heap**

```
1. class Collar { }
2.
3. class Dog {
4.     Collar c; // variabilă instanță
5.     String name; // variabilă instanță
6.
7.     public static void main(String [] args) {
8.
9.         Dog d; // variabilă locală
10.        d = new Dog();
11.        d.go(d);
12.    }
13.    void go(Dog dog) { // variabilă locală: dog
14.        c = new Collar();
15.        dog.setName("Aiko");
16.    }
17.    void setName(String dogName) { // variabilă locală: dogName
18.        name = dogName;
19.        // alte lucruri
20.    }
21. }
```



# Alocare memorie pe **stack** și **heap**





# Alocare memorie pe **stack** și **heap**

- Linia 7: metoda `main()` este plasată pe stivă
- Linia 9: variabila referință `d` este creată pe stivă (dar încă nu este nici un obiect de tip Dog)
- Linia 10: un obiect de tipul Dog este creat pe în memoria heap; variabila `d` este o referință către acest obiect
- Linia 11: o copie a variabilei referință `d` este transmisă ca argument la apelul metodei `go()`
- Linia 13: metoda `go()` este plasată pe stivă cu parametrul `dog` ca variabilă locală
- Linia 14: un nou obiect Collar este creat în memoria heap
- Linia 17: `setName()` este adăugată pe stivă cu parametrul `dogName` ca variabilă locală

Computer Science



# Alocare memorie pe **stack** și **heap**

- Linia 18: variabila instanță name referă acum către obiectul `dogName`
- După terminarea liniei 19, `setName()` este finalizată și este scoasă de pe stivă împreună cu toate variabilele sale locale (`dogName`). Doar obiectul String "Aiko" rămâne deoarece mai există o referință către el
- După terminarea liniei 15, metoda `go()` este scoasă de pe stivă. Obiectul String "Aiko" nu mai este referit
- După terminarea liniei 11, metoda `main` este scoasă de pe stivă. Obiectul d nu mai este referit. Execuția programului se termină



# Reguli pentru proiectarea claselor

- *Inițializați întotdeauna datele*
  - Java nu va inițializa variabilele locale, dar va inițializa variabilele instanță ale obiectelor
  - Nu vă bazați pe valorile implicite, ci inițializați variabilele explicit
- *Nu folosiți prea multe tipuri într-o clasă*
  - Înlocuiți folosirile multiple *înrudite* ale tipurilor de bază cu alte clase.  
Spre exemplu:

```
private String strada;  
private String oras;  
private String stat;  
private String tara;  
private int codPostal;
```



```
class Adresa {  
    private String strada;  
    private String oras;  
    private String stat;  
    private String tara;  
    private int codPostal;  
}
```



# Reguli pentru proiectarea claselor

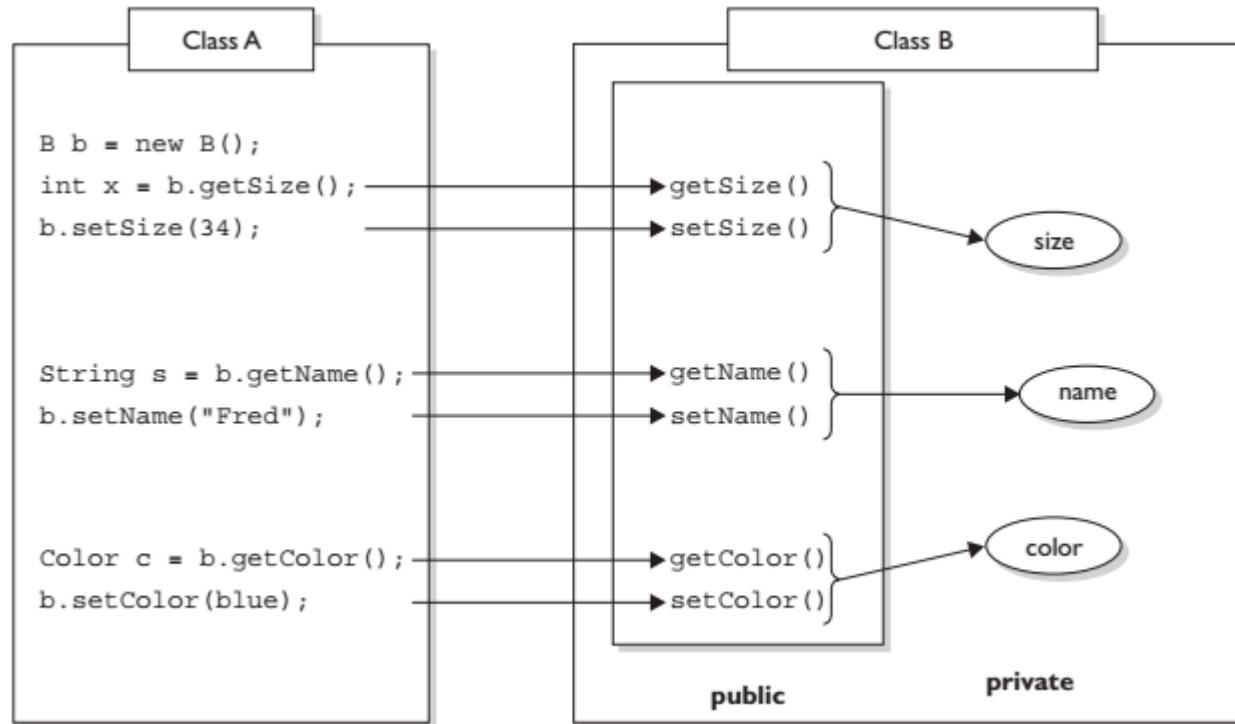
## ■ Folosiți Încapsularea

- Păstrați datele întotdeauna private
  - Schimbările în reprezentarea datelor nu vor afecta utilizatorii claselor; erorile sunt mai ușor de detectat
- Pentru accesul sau modificarea valorilor variabilelor instantă se vor implementa metode *accesori* și *mutatori* (*getter* și *setter*)
- Nu toate câmpurile necesită accesori și mutatori individuali
  - Exemplu: Angajat – obține și setează salariul, dar nu și data angajării (nu se schimbă o dată creată)



# Reguli pentru proiectarea claselor

- Exemplu de cod pentru încapsulare:



- Clasa A nu poate accesa variabilele instanță ale clasei B decât prin intermediul metodelor getter și setter



# Reguli pentru proiectarea claselor

## ■ Contra-exemplu de cod pentru încapsulare:

```
class Foo {  
    public int left = 9;  
    public int right = 3;  
    public void setLeft(int leftNum) {  
        left = leftNum;  
        right = leftNum/3;  
    }  
}
```

- Întrebare: Variabila right va fi totdeauna  $1/3$  din left?
- Răspuns: Nu, deoarece utilizatorii clasei Foo pot seta direct valoarea variabilelor left și right, fără să apeleze metoda setLeft()



# Reguli pentru proiectarea claselor

- Folosiți o formă standard pentru definirea claselor și, pentru fiecare secțiune, scrieți în ordine
  - *constante*
  - *constructori*
  - *metode*
  - *metode statice*
  - variabile instanță
  - variabile statice
- De ce? Utilizatorii sunt mai interesați de interfața publică decât de datele private și mai mult de metode decât de date



# Reguli pentru proiectarea claselor

- Divizați clasele cu prea multe responsabilități, de exemplu

```
class PachetCarti { // nu este recomandat
    public PachetCarti() { . . . }
    public void amesteca() { . . . }
    public void obtineValoareaMaxima() { . . . }
    public void obțineFelulMaxim() { . . . }
    public void rangulMaxim() { . . . }
    public void deseneaza() { . . . }

    private int[] valoare;
    private int[] fel;
    private int carti;
}

// creați clasa Carte!!!
```



# Reguli pentru proiectarea claselor

- Faceti numele claselor si metodelor sa reflecte responsabilitatile acestora
- O convenție bună este:
  - Numele clasei: substantiv (ex. Comanda) sau substantiv+adjectiv (ex. ComandaUrgenta)
  - Numele metodelor: verbe; accesoriile încep cu "get"; mutatorii încep cu "set"

Computer Science



# O clasă "Complex" foarte simplă

```
public class ComplexTrivial
{
    //Proprietăți
    private double real;
    private double img;
    // Constructor care
    // initializează valorile
    public ComplexTrivial (double r, double i)
    { real = r; img = i; }
    // Definește o metodă pentru adunare
    public void aduna(ComplexTrivial cvalue) {
        real = real + cvalue.real;
        img = img + cvalue.img;
    }
    // Definește o metodă pentru scădere
    public void scade(ComplexTrivial cvalue) {
        real = real - cvalue.real;
        img = img - cvalue.img;
    }
}
```

## ComplexTrivial

-real: double  
-img: double

<<constructor>>+ComplexTrivial(r: double, i: double)  
<<mutator>>+aduna(cvalue: ComplexTrivial)  
<<mutator>>+scade(cvalue: ComplexTrivial)



# O clasă "Comutator" simplă

```
// Un comutator on/off simplu

class ComutatorSimplu {

    // Comută pe închis.
    public void inchide() {
        System.out.println("Inchide
                            comutatorul");
        seteazaInchis(true);
    }

    // Comută pe deschis.
    public void deschide() {
        System.out.println(" Deschide
                            comutatorul");
        seteazaInchis(false);
    }

    // Raportează dacă comutatorul
    // este închis sau nu.
    public boolean esteInchis () {
        return obtineStareInchis();
    }

    // Returnează starea comutatorului
    private boolean obtineStareInchis()
    { return inchis; }

    // Setează starea comutatorului.
    private void seteazaInchis(boolean o)
    {
        inchis = o;
    }

    // Dacă comutatorul este închis sau nu
    // true inseamnă închis, fals inseamnă
    // deschis.
    private boolean inchis = false;
}
```

- Obs. **javadoc** nu va genera o documentație corespunzătoare pentru acest mod de comentare



# Tipuri de clase în Java

- O *clasă de nivel maxim* (top level class) nu apare în interiorul altei clase sau interfețe
- Dacă un tip nu este de nivel maxim atunci este *imbricat* (nested)
  - Un tip poate fi *membru* al altui tip
  - Un tip membru este inclus direct într-o altă declarație de tip
  - Unii membrii sunt *clase interioare* (inner classes) și cuprind:
    - *Clase locale*: clase cu nume declarate înăuntrul unui bloc (corpul unui constructor sau al unei metode)
    - *Clase anonte*: clase nenumite ale căror instanțe sunt create în expresii sau în instrucțiuni

Computer Science



# Clase interioare (interne)

- *Clasă interioară*: clasă definită înăuntrul alteia
  - Permite *gruparea claselor* care țin logic una de alta pentru a *controla vizibilitatea* uneia în cealaltă
  - Clasele interne sunt diferite de compozitie
- Crearea unei instanțe de clasă internă se face obișnuit de oriunde
  - Excepție fac metodele statice ale clasei exterioare când se specifică tipul obiectului ca  
*NumeleClaseiExterioare.NumeleClaseiInterne*



# Exemplu de clasă internă

- Tipic, o clasă externă va avea o metodă care returnează o referință la clasa internă

```
public class Parcel {  
    class Contents {  
        private int val = 10;  
        public int value() {return val;}  
    }  
    class Destination {  
        private String label;  
        Destination(String dst) {  
            label = dst;  
        }  
        String readLabel() {return label;}  
    }  
  
    public Destination to(String s) {  
        return new Destination(s);  
    }  
}
```

```
public Contents cont() {  
    return new Contents();  
}  
public void ship(String dest) {  
    Contents c = cont();  
    Destination d = to(dest);  
    System.out.println(d.readLabel());  
}  
public static void main(String[] args){  
    Parcel p = new Parcel();  
    p.ship("Romania");  
    Parcel q = new Parcel();  
    // Definire de referințe către clase  
    // interne:  
    Parcel.Contents c = q.cont();  
    Parcel.Destination d = q.to("China");  
}
```

Computer Science



# Clase interne

- De patru feluri:
  - Clase membru statice
  - Clase membru
  - Clase locale
  - Clase anonte
- Clasă *membru statică* este un membru static al unei clase
  - Are acces la toate metodele *statice* ale clasei exterioare

```
class MyOuter {  
    public static class MyInner {  
        //...  
        public void metoda1() {}  
        //alte metode  
    }  
}
```



# Clase interne

- *Clasă membru* : definită și ea ca membru al clasei
  - Este *specifică instanței* și
  - Are *acces la toate* metodele și membrii, chiar și la referința *this* a clasei exterioare

```
class MyOuter {  
    private float variable = 0;  
    public void doSomething() { ... }  
    private class MyInner {  
        public void doSomething() { ... }  
        public void method() {  
            //Apelul unei metode cu același nume a clasei exterioare  
            MyOuter.this.doSomething();  
        }  
    }  
}
```



# Clase interne

- *Clasele locale*: declarate într-un bloc de cod; vizibile în acel bloc, ca orice altă metodă/variabilă

```
interface MyInterface {  
    public String getInfo();  
}  
class MyOuter {  
    MyInterface current_object;  
    public void setInterface(String info) {  
        class MyInner implements MyInterface {  
            private String info;  
            public MyInner(String inf) {info=inf;}  
            public String getInfo() {return info;}  
        }  
        current_object = new MyInner(info);  
    }  
}
```



# Clase interne

- Clasă *anonomă*: este o clasă locală fără nume
  - Exemplu:

```
MyAnonymousClass frenchGreeting = new MyAnonymousClass () {  
    private String info;  
    public MyAnonymousClass(String inf) {info=inf;}  
    public String getInfo() {return info;}  
};
```
- Sunt foarte des folosite la implementarea interfețelor grafice când este necesar să se adauge ascultători anumitor anumitor componente

Computer Science



# Elemente de bază despre tablouri

- În Java, un *tablou* este o colecție indexată de date de același tip
- Tablourile sunt utile la sortări și la manipularea unei colecții de valori
- În Java, un tablou este un tip de dată referință
- Se folosește operatorul **new** pentru a aloca memorie pentru stocarea valorilor într-un tablou

```
//creează un tablou de mărime 12
double[] precipitatii;
precipitatii = new double [12];
```

- Folosim o *expresie indexată* pentru a ne referi la elemente individuale din colecție
- Tablourile folosesc indexarea de la zero



# Elemente de bază despre tablouri

- Un tablou are o *constantă publică length* care conține dimensiunea tabloului
- Nu confundați *valoarea length* a unui tablou cu *metoda length()* a unui obiect **String**
- **length()** este metodă pentru un obiect **String**, deci folosim sintaxa pentru metodă  

```
String str = "acesta este un sir";  
int size = str.length();
```
- Pe de altă parte, un tablou este un tip de dată referință, nu un obiect. De aceea nu folosim apel de metodă:  

```
int size = precipitatii.length;
```



# Elemente de bază despre tablouri

- Folosirea constantelor pentru declararea dimensiunilor tablourilor nu duce întotdeauna la folosirea eficientă a spațiului
- Declararea cu dimensiuni fixe poate pune două probleme:
  - Capacitatea poate fi insuficientă pentru sarcina de îndeplinit
  - Spațiu irosit
- Java, nu este limitat la declararea cu dimensiune fixă
- După creare însă, un tablou este o structură de lungime fixă
- Indiferent de tipul de tablou cu care se lucrează, variabila tablou este o referință la un obiect creat pe heap
- Accesul la datele din tablou se fac prin acest obiect tablou
- Obiectul tablou are o variabilă *identificator* unică



# Elemente de bază despre tablouri

- Codul următor cere utilizatorului dimensiunea unui tablou și declară un tablou de dimensiunea cerută:

```
int size;  
int[] number;  
size= Integer.parseInt(JOptionPane.showInputDialog(null,  
        "Marimea tabloului:"));  
number = new int[size];
```

- Orice expresie aritmetică validă este permisă la specificarea dimensiunii unui tablou:

```
size = Integer.parseInt(  
        JOptionPane.showInputDialog(null, ""));  
number = new int[size*size + 2* size + 5];
```

- Tablourile nu sunt limitate la tipurile de date primitive



# Tablourile sunt variabile referință

```
int[] data;
```

*data este o variabilă referință al cărei tip este `int[]`, însemnând "tablou de int". În acest moment valoarea sa este null.*

```
data = new int[5];
```

*Operatorul `new` face să se aloce pe heap o zonă de memorie destul de mare pentru 5 int. Aici, lui `data` i se asignează o referință la adresa din heap.*

```
data[0] = 6;  
data[2] = 12;
```

*Inițial, toți cei cinci int sunt 0. Aici, la doi dintre ei li se atribuie alte valori.*

```
int[] info = {6, 10, 12, 0, 0};
```

```
int[] info = new int[]{6, 10, 12, 0, 0};
```



# Excepții de depășire a limitelor tablourilor

```
public class ArrayTool{  
    public int sum(int[] data) {  
        int sum = 0;  
        for (int i = 0; i < data.length; i++) {  
            sum += data[i];  
        }  
        return sum;  
    }  
    public int sum2(int[] data) {  
        int sum = 0;  
        for (int i = 0; i <= data.length; i++) {  
            sum += data[i];  
        }  
        return sum;  
    }  
}
```

Folosirea acestei comparații produce aruncarea unei excepții **ArrayIndexOutOfBoundsException**



# Tablouri de tipuri primitive

```
int[] data;
```

data      null

```
data = new int[3];
```

data      500      →      0      0      0

```
data[0] = 5;
```

```
data[1] = 10;
```

data      500      →      5      10      0



# Tablouri de alte tipuri primitive

```
double[] temps;
temp = new double[24];
temp[0] = 18.5;
temp[1] = 24.2;

boolean[] raspunsuri = new boolean[6];
. . .
if (raspunsuri[0])
    faCeva();

char[] pfile = new char[500];
deschide un fișier pentru citire
while (mai sunt caractere în fișier & pFile nu este plin)
    pfile[i++] = caracter din fișier
```



# Tablouri bidimensionale

- Tablourile pot avea 2, 3, sau mai multe dimensiuni
- La declararea unei variabile pentru un astfel de tablou, folosiți câte o pereche de paranteze pătrate pentru fiecare dimensiune
- Pentru tablourile bidimensionale, elementele sunt indexate [rind][coloana]
- Exemplu:

```
char[][] tabla;  
  
tabla = new char[3][3];  
  
tabla[1][1] = 'X';  
  
tabla[0][0] = 'O';  
  
tabla[0][1] = 'X';
```



# Tablouri de obiecte

## O clasă contor:

```
public class Counter {  
    private int numar;  
  
    /**  
     * Constructor. Initializeaza  
     * contorul la zero.  
     */  
    public Counter() {  
        numar = 0;  
    }  
  
    /**  
     * @return valoarea curenta a  
     * contorului  
     */  
    public int obtineNumar() {  
        return numar;  
    }  
}
```

```
/**  
 * Incrementeaza contorul  
 * cu unu  
 */  
public void increment() {  
    numar++;  
}  
  
/**  
 * Reseteaza contorul  
 * la zero  
 */  
public void reset() {  
    numar = 0;  
}
```



# Tablouri de obiecte

```
Counter[] counters;
```

counters null

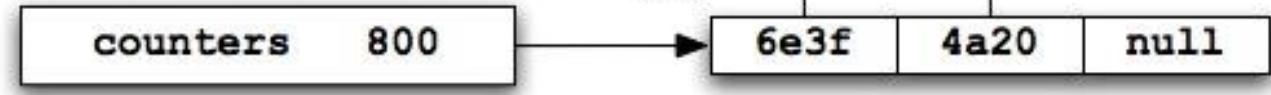
STACK

HEAP

```
counters = new Counter[3];
```



```
counters[0]=new Counter();
counters[0].increment();
counters[1]=new Counter();
```



STACK

HEAP



# Exemplu: Un joc Tic-Tac-Toe

```
public class TicTacToe{  
    // Variabile instanță  
    /* tablou bidimensional de caractere pentru  
     * tabla  
     */  
    private char[][] tabla;  
  
    /** Constructor – crează o tablă în care  
     * fiecare pătrat conține un caracter  
     * sublinieră '_'.  
     */  
    public TicTacToe() {  
        tabla = new char[3][3];  
        for (int rind = 0; rind < 3; rind++) {  
            for (int col = 0; col < 3; col++) {  
                tabla[rind][col] = '_';  
            } // sfârșit bucla interioară  
        } // sfârșit bucla exterioară  
    }  
}
```

```
    /** Pune caracterul c pe poziția [rind][col] a  
     * tablei de joc dacă rind, col și c sunt valide  
     * și pătratului de la [rind][col] nu i-a fost  
     * atribuită deja o valoare ( alta decât valoarea  
     * implicită, '_').  
     * @param rind rindul de pe tabla  
     * @param col coloana de pe tabla  
     * @param c caracter folosit la marcarea  
     * @return true dacă reușește, alfel false  
     */  
    public boolean set(int rind, int col, char c) {  
        if (rind >= 3 || rind < 0)  
            return false;  
        if (col >= 3 || col < 0)  
            return false;  
        if (tabla[rind][col] != '_')  
            return false;  
        if ( !(c == 'X' || c == 'O'))  
            return false;  
        // assertiune: rind, col, c sunt valide  
        tabla[rind][col] = c;  
        return true;  
    }  
}
```



# Exemplu: Un joc Tic-Tac-Toe

```
/**  
 * @return caracterul din poziția [rind][col] de pe tabla.  
 * @param rind rindul de pe tabla  
 * @param col coloana de pe tabla  
 */  
public char get(int rind, int col) {  
    return tabla[rind][col];  
}  
  
/** Tipărește starea tablei, d.e.  
 *  
 *      _ _ _  
 *      | X O  
 *      | O _ X  
 */  
public void print(){  
    for (int rind = 0; rind < 3; rind ++){  
        for (int col = 0; col < 3; col++){  
            System.out.print(tabla[rind][col] + " ");  
        } // sfârșit bucla interioară  
        System.out.println();  
    } // sfârșit bucla exterioară  
}
```

## Exerciții:

- Completați jocul pentru a-l face jucabil (poate mai definiți și alte clase?)
- Modificați clasa astfel încât să permită dimensiuni mai mari ale tablei de joc



# Prescurtări la inițializarea tablourilor

Tablouri de tipuri primitive:

```
int[] info1 = { 2000, 100, 40, 60};  
int[] info2 = new int[]{ 2000, 100, 40, 60};  
char[] choices1 = { 'p', 's', 'q'};  
char[] choices2 = new char[]{ 'p', 's', 'q'};  
double[] temps1 = {75.6, 99.4, 86.7};  
double[] temps2 = new double[] {75.6, 99.4, 86.7};
```

Computer Science



# Prescurtări la inițializarea tablourilor

Tablouri de obiecte:

```
Person[] people = {new Person("jo"), new Person("flo")};  
Person[] people = new Person[] {new Person("jo"),  
                                new Person("flo")};  
  
Point p1 = new Point(0,0);  
Point[] points1 = {p1, new Point(0, 10)};  
Point[] points2 = new Point[] {p1, new Point(0, 10)};
```

**Observație: Construcția sintactică “new type[ ]” poate fi folosită la o asignare care nu este și o declarație de variabilă**



# Transmiterea tablourilor ca parametri

- Atunci când nu mai există nici o referință spre un obiect, sistemul va șterge obiectul și va elibera memoria ocupată de acesta
  - Ștergerea unui obiect se numește *deallocarea* memoriei
  - Procesul de dealocare a memoriei se numește *colectarea reziduurilor* și este realizat automat în Java
- La transmiterea ca *parametru* a unui *tablou* spre o metodă, se transmite doar o *referință* spre tabloul respectiv
  - Nu se creează o copie a tabloului în metodă



# Capcană: un tablou de caractere nu este un String

- Un tablou de caractere nu este un obiect de clasă **String**

```
char[] a = {'A', 'B', 'C'};  
String s = a; //Illegal!
```
- Un tablou de caractere poate fi convertit la un obiect de clasă **String**
- Un tablou de caractere este conceptual o listă de caractere și de aceea este conceptual ca un sir
- Clasa **String** are un constructor cu un singur parametru de tip **char[]**

```
String s = new String(a);
```

- Obiectul **s** va avea aceeași secvență de caractere ca întregul tablou **a** ("ABC"), dar este o copie *independentă*



# Capcană: un tablou de caractere nu este un String

- Un alt constructor al **String** folosește o subgamă a tabloului de caractere
  - Fiind dat **a** ca mai înainte, noul obiect String este "**AB**"
- Un tablou de caractere are ceva în comun cu obiectele **String**
  - Un tablou de caractere poate fi tipărit folosind **println**  
**System.out.println(a);**
  - Dat fiind **a** ca mai înainte, se va tipări  
**ABC**



# Copierea tablourilor

- Clasa **System** are o metodă numită **arraycopy**

- Folosită la copierea eficientă a datelor între tablouri

```
public static void
    arraycopy(
        Object src,
        int srcPos,
        Object dest,
        int destPos,
        int length
    )
```

```
import java.lang.*;
public class SystemDemo {
    public static void main(String[] args) {
        // Declară două tablouri:
        int[] arr1 = {1,2,3,4,5,6};
        int[] arr2 = {0,2,4,6,8,10};

        /* Copiază două elemente din arr1 începând
           cu cel de-al doilea element în arr2
           începând cu cel de-al patrulea
           element: */
        System.arraycopy(arr1, 1, arr2, 3, 2);

        // Afisează conținutul lui arr2:
        for (int i=0; i<arr2.length; i++)
            System.out.println("Elementul #" + i +
                               " = " + arr2[i]);
    }
}
```



# Programare orientată pe obiecte

1. Pachete (*packages*)
2. Moștenire



# Organizarea claselor înrudite în pachete

- Pachet (package): set de clase înrudite
- Pentru a pune o clasă într-un pachet, trebuie scrisă o astfel de linie

```
package numePachet;
```

ca primă instrucțiune în fișierul sursă care conține clasa

- Numele pachetului constă din unul sau mai mulți identificatori separați prin puncte



# Organizarea claselor înrudite în pachete

- Spre exemplu, pentru a pune clasa **Database** într-un pachet numit **oop.examples**, fișierul **Database.java** trebuie să înceapă astfel:

```
package oop.examples;  
public class Database  
{  
    . . .  
}
```

- Pachetul implicit nu are nume, deci nu are o specificare **package**



# Organizarea claselor înrudite în pachete

Pachet	Scop	Exemplu de clasă
java.lang	suport pentru limbaj	Math
java.util	utilitare	Random
java.io	intrare și ieșire	PrintScreen
java.awt	Abstract Windowing Toolkit	Color
java.applet	Applets	Applet
java.net	Networking	Socket
java.sql	accesul la baze de date	ResultSet
java.swing	interfața utilizator swing	JButton
org.omg.CORBA	Common Object Request Broker Architecture	IntHolder



# Importul pachetelor

- Se poate folosi întotdeauna o clasă fără import

```
java.util.Scanner s = new java.util.Scanner(System.in);
```

- Dar e greoi să folosim nume calificate complet

- „import” ne permite să folosim nume mai scurte pentru clase

```
import java.util.Scanner;
```

```
. . .
```

```
Scanner in = new Scanner(System.in)
```

- Putem importa toate clasele dintr-un pachet

```
import java.util.*;
```

- Nu este nevoie să importăm `java.lang`

- Nu este nevoie să importăm alte clase din același pachet



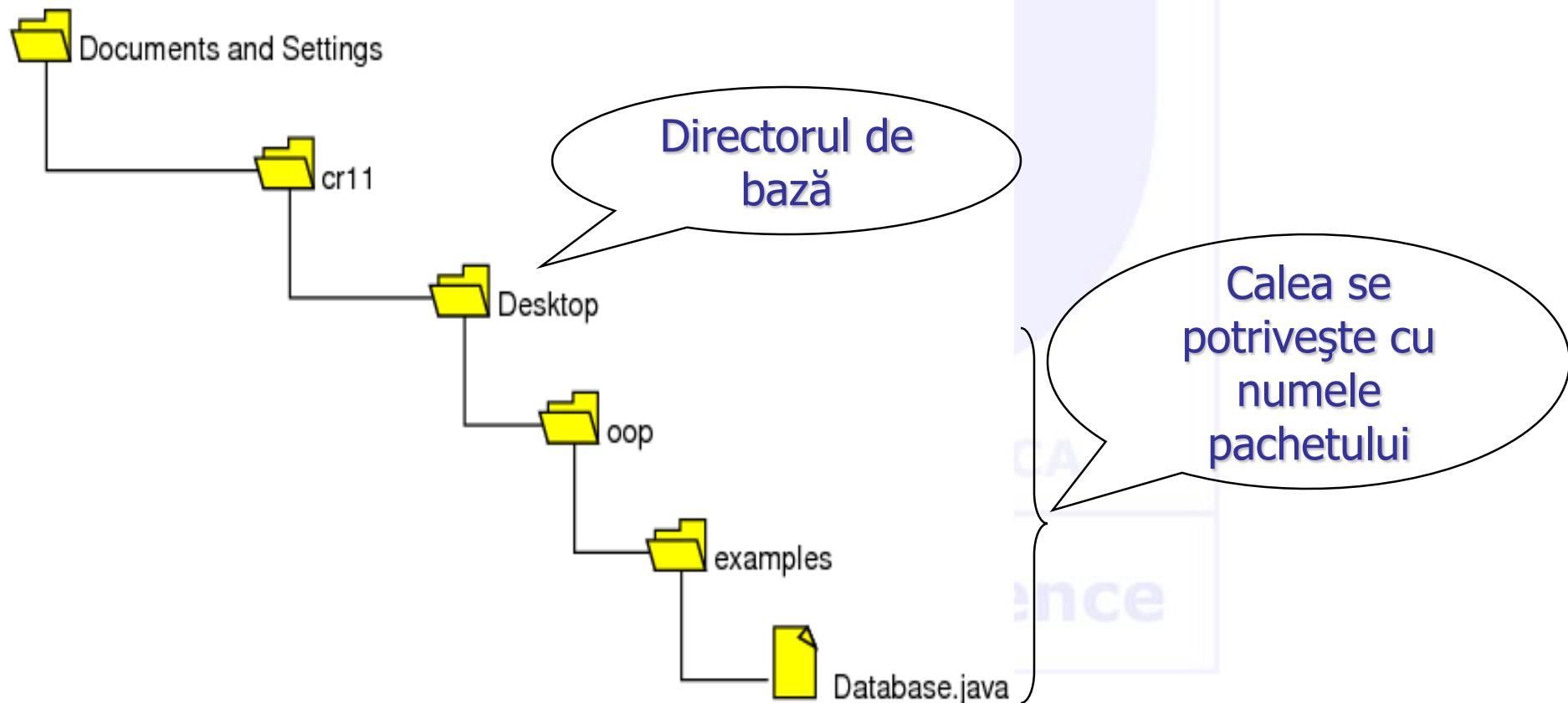
# Nume de pachete și determinarea locului unde se află clasele

- Folosiți pachete pentru a evita conflictele de nume (două clase diferite având același nume – Timer – situate în două pachete diferite)  
`java.util.Timer` vs. `javax.swing.Timer`
- Numele de pachete trebuie să fie neambigue
- Numele căii trebuie să se potrivească cu numele pachetului  
`oop/examples/Database.java`
  - Calea spre clase conține directoarele de bază care pot conține directoare de pachet



# Directoare de bază și subdirectoare pentru pachete

```
set CLASSPATH=C:\Documents and Settings\cr11\Desktop; .
```





# Cum se construiește un pachet

1) Puneti o linie cu numele pachetului la inceputul fiecarei clase.

```
package pachetDulciuri;  
public class Ciocolata {  
    . . .  
}                                package pachetDulciuri;  
public class Jeleu {  
    . . .  
}                                package pachetDulciuri;  
public class Drops {  
    . . .  
}
```

2) Stocați fișierele Java din pachet într-un director comun.

pachetDulciuri

Ciocolata.java

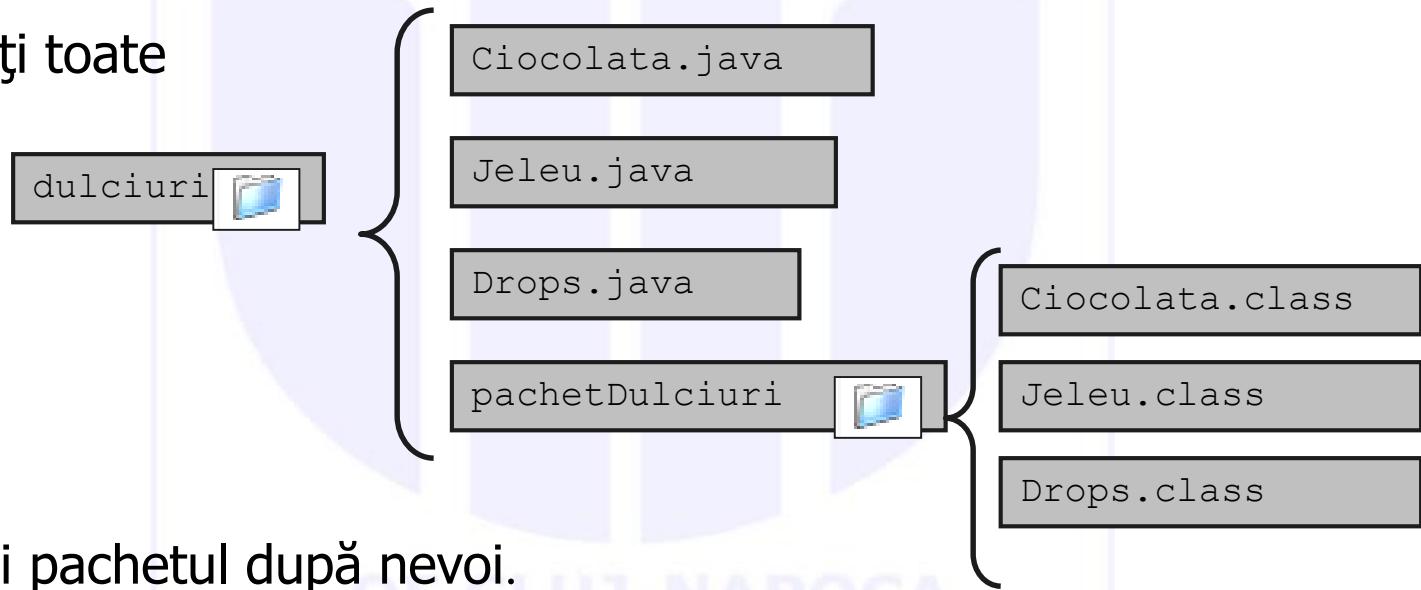
Jeleu.java

Drops.java



# Cum se construiește un pachet

3) Comparați toate fișierele.



4) Importați pachetul după nevoi.

```
import dulciuri.pachetDulciuri.*;  
public class ConsumatorDulciuri { . . . }
```



# Cum să refolosim codul?

- Putem scrie clase de la început – fără a refolosi nimic (o extremă!)
  - Ceea ce unii programatori doresc să facă întotdeauna
- Putem găsi o clasa existentă care se potrivește exact cerințelor problemei (o altă extremă!)
  - Cel mai ușor lucru pentru programator
- Putem construi clase din clase existente bine testate și bine documentate
  - Un fel de refolosire foarte tipic, numit refolosire prin **compoziție!**
- Putem refolosi o clasă existentă prin **moștenire**
  - Necesară mai multe cunoștințe decât refolosirea prin compozitie



# Moștenirea

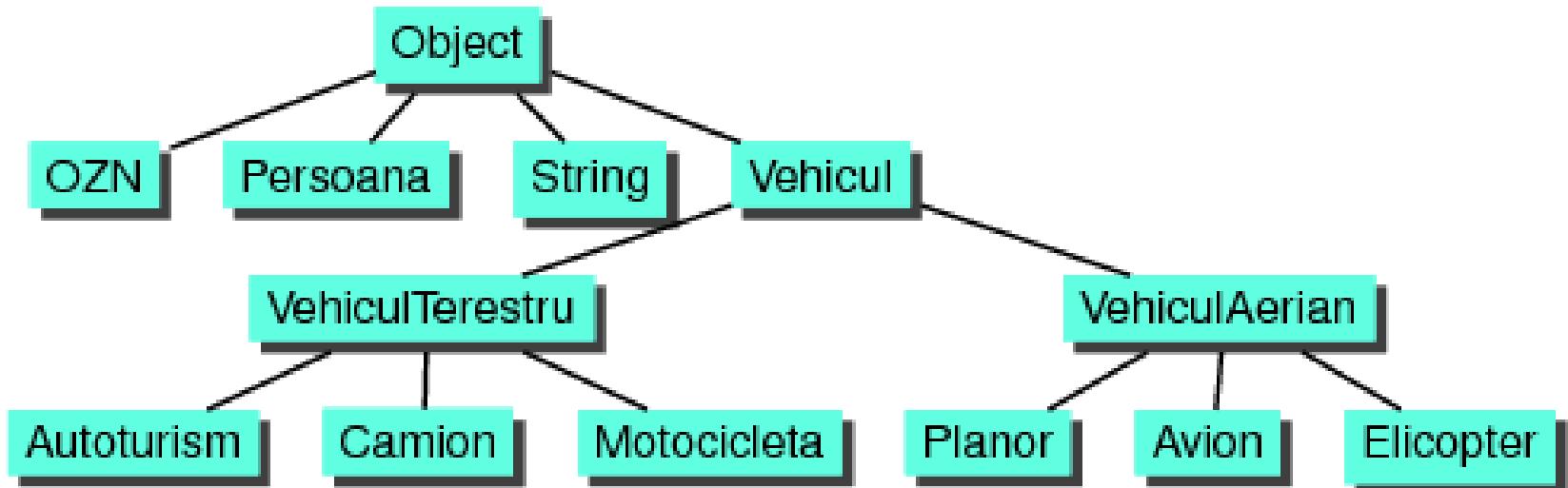
- **Moștenirea** este una din tehniciile principale ale programării orientate pe obiecte
- Folosind această tehnică:
  - se definește mai întâi o formă foarte generală de clasă și se compilează, apoi
  - se definesc versiuni mai specializate ale clasei prin adăugarea de variabile instanță și de metode
- Despre clasele specializate se spune că *moștenesc* metodele și variabilele instanță ale clasei generale

Computer Science



# Moștenirea

- Moștenirea modelează relații de tipul “*este o(un)*”
  - Un obiect “este un” alt obiect dacă se poate comporta în același fel
  - Moștenirea folosește *asemănările și deosebirile* pentru a modela grupuri de obiecte înrudite
- Unde există moștenire, există și o *ierarhie de moștenire* a claselor





# Moștenirea

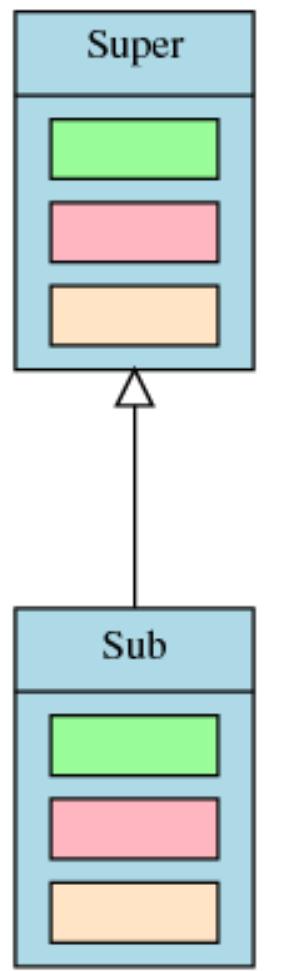
- Moștenirea este un mod de:
  - organizare a informației
  - grupare a claselor similare
  - modelare a asemănărilor între clase
  - creare a unei taxonomii de obiecte
- **Vehicul** este numit *superclasă*
  - sau *clasă de bază* sau *clasă părinte*
- **VehiculTerestru** este numit *subclasă*
  - sau *clasă derivată* sau *clasa fiică*
- Oricare clasă poate fi de ambele feluri în același timp
  - D.e., **VehiculTerestru** este superclasă pentru **Camion** și subclasă pentru **Vehicul**



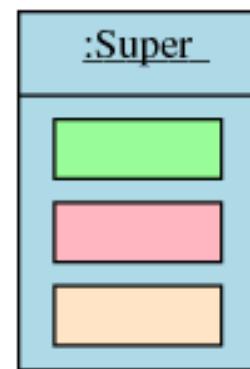
# Moștenirea

- În Java fiecare clasă extinde clasa **Object** fie direct, fie indirect
- O clasă are în mod automat toate variabilele instanță și metodele clasei de bază și poate avea și metode suplimentare și/sau variabile instanță
- Moștenirea este avantajoasă deoarece permite să se *refolosească* codul, fără a fi nevoie să fie copiat în definițiile claselor derivate
- În Java se poate moșteni de la *o singură* superclasă
  - Nu există limite pentru adâncimea sau lățimea ierarhiei de clase

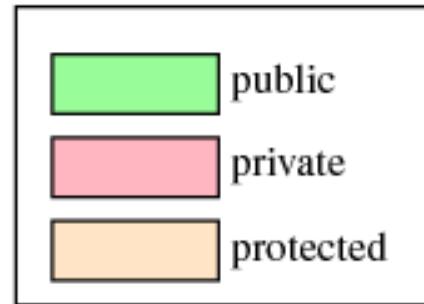
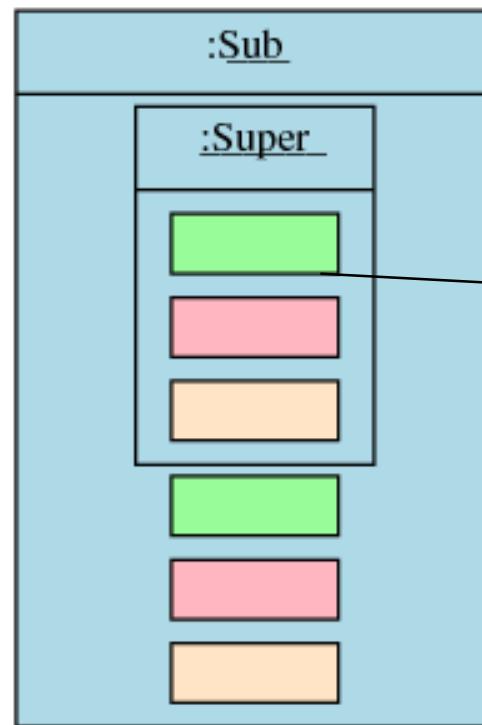
Computer Science



Ierarhia de clase



Instanțe  
POO4 - T.U. Cluj

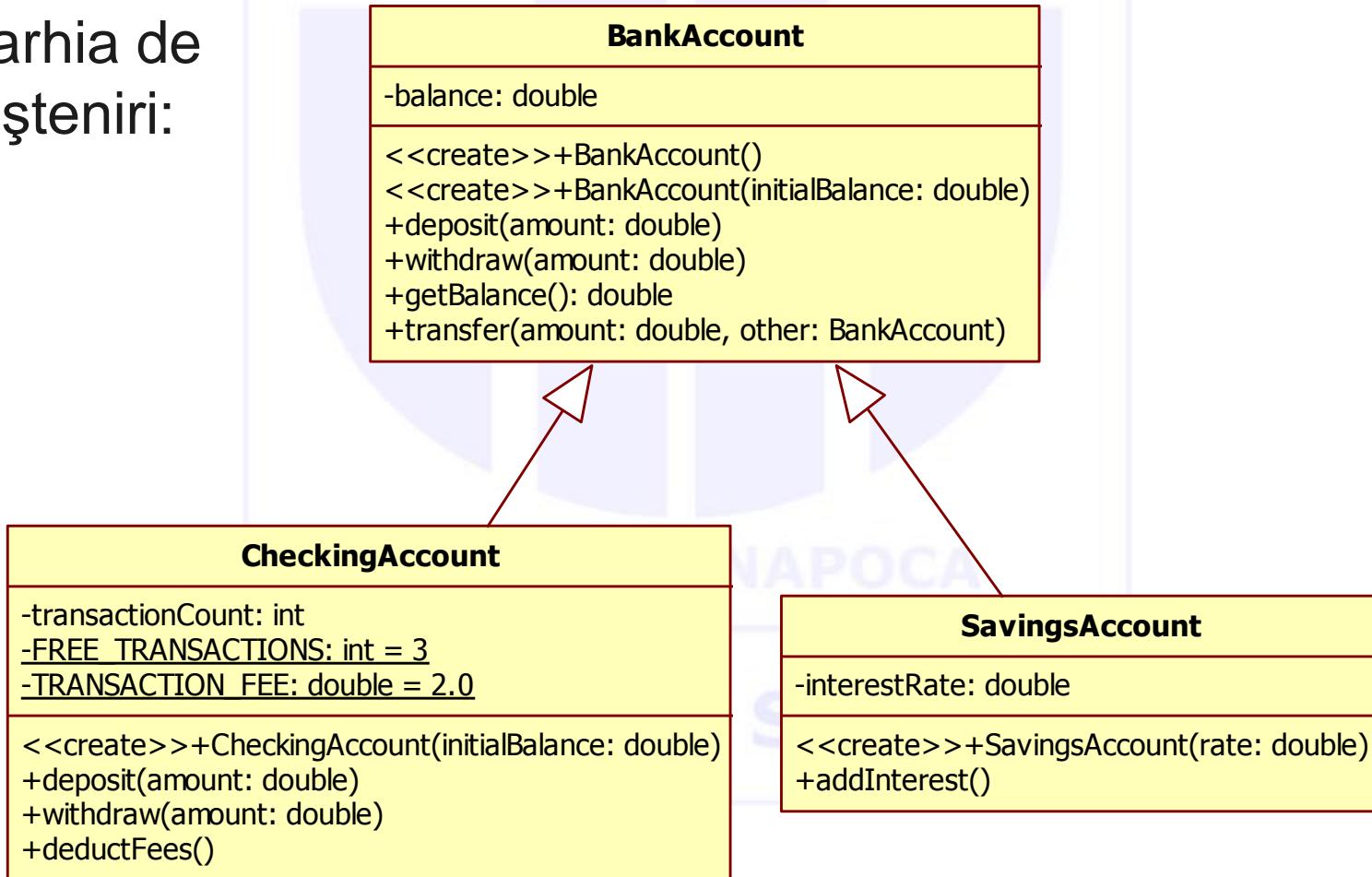


**Componentele moștenite ale superclasei sunt parte a subclasei**



# Exemplu: ierarhia unor conturi bancare

- Ierarhia de moșteniri:





# Exemplu: ierarhia unor conturi bancare

- Scurtă specificație
  - Toate conturile bancare suportă metoda `getBalance` – obține soldul contului
  - Toate conturile bancare suportă metodele `deposit` (depune) și `withdraw` (retrage), dar implementările diferă
  - Contul de cecuri (`CheckingAccount`) are nevoie de o metodă pentru deducerea taxelor de prelucrare – `deductFees`; contul de economii (`SavingsAccount`) are nevoie de o metodă pentru adăugarea dobânzii – `addInterest`



# Clase derivate

- Cum un cont de economii este un cont bancar, el este definit ca o clasă *derivată* a clasei **BankAccount**
  - O clasă *derivată* se definește prin adăugarea de variabile și/sau metode la o clasă existentă
  - Fraza **extends BaseClass** trebuie adăugată în definiția clasei derivate:

```
public class SavingsAccount extends BankAccount
```
- Sintaxa pentru moștenire:

```
class NumeSubclasa extends NumeSuperclasa
{
    metode
    câmpuri de instanță
}
```



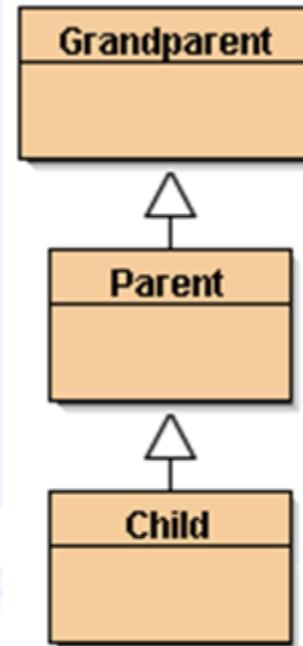
# Clase derivate (subclase)

- O clasă derivată, numită și *subclasă*, este definită pornind de la o altă clasă definită deja, numită *clăsă de bază* sau *superclasă*, prin adăugarea (și/sau modificarea) de metode, variabile instanță și de variabile statice
  - Clasa derivată *moștenește* toate *metodele*, toate *variabilele instanță*, precum și toate *variabilele statice* din clasa de bază
  - Clasa derivată *poate adăuga* variabile instanță, variabile statice și/sau metode
- *Definițiile* variabilelor și metodelor moștenite *nu apar* în clasa derivată
  - Codul este reutilizat fără a fi nevoie să fie copiat explicit, cu excepția cazului în care creatorul clasei derivate nu *redefineste* una sau mai multe dintre *metodele* clasei



# Clase părinti și clase copii

- O clasă de bază este numită adesea *clasă părinte*
  - Clasa derivată se mai numește și *clasă fiică (copil)*
- Aceste relații sunt adesea extinse astfel că o clasă este părintele unui părinte al unei alte clase și se numește *clasă strămoș*
  - Dacă clasa **Grandparent** este un strămoș al clasei **Child**, atunci clasa **Child** poate fi numită clasă *descendentă* a clasei **Grandparent**





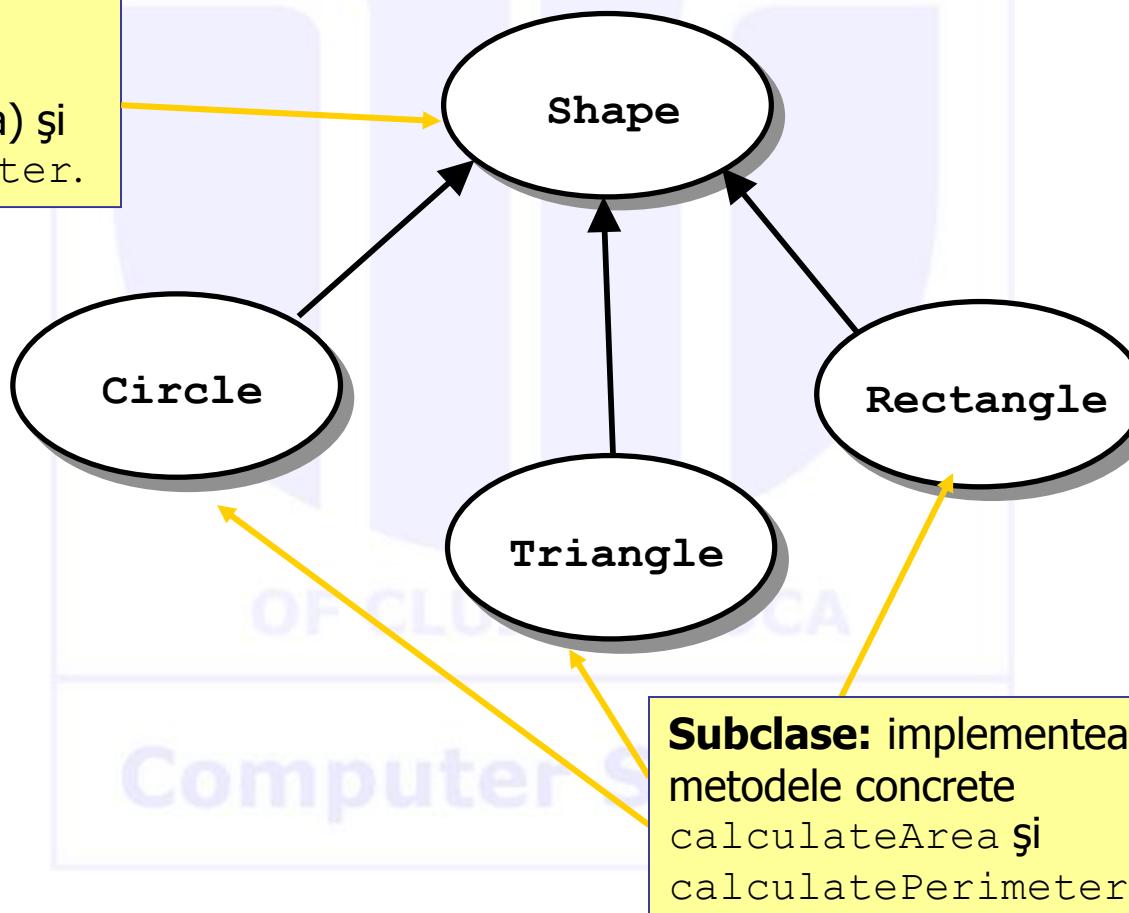
# Clase abstracte

- O metodă sau o clasă abstractă se declară folosind cuvântul cheie **abstract**
- O clasă care conține cel puțin o metodă abstractă trebuie să fie abstractă
- Dintr-o clasă abstractă nu se poate instanția nici un obiect
- Fiecare subclasă a unei clase abstracte care va fi folosită pentru a instanția obiecte trebuie să ofere implementări pentru toate metodele abstracte din superclasa
- Clasele abstracte economisesc timp, deoarece nu trebuie să scriem cod “inutil” care n-ar fi executat niciodată
- O clasă abstractă poate moșteni metode *abstracte*
  - dintr-o interfață sau
  - dintr-o clasă



# Exemplu: O clasă numită Shape (formă)

**Superclasă:** conține metodele abstracte calculateArea (calculează suprafața) și calculatePerimeter.



**Subclase:** implementează metodele concrete calculateArea și calculatePerimeter.



# Exemplu: O clasă numită Shape

```
/**  
 * Abstract class Shape - base for inheritance for shapes  
 */  
  
public abstract class Shape {  
    private static int counter;  
    // Constructor  
    public Shape() {  
        counter++;  
    }  
    // calculate area  
    public abstract double calculateArea();  
    // calculate perimeter  
    public abstract double calculatePerimeter();  
    // get number of shapes  
    public int getCount() {  
        return counter;  
    }  
    protected void finalize() throws Throwable {  
        counter--;  
    }  
}
```

## Definiția superclasei.

Observați că această clasă este declarată abstract.

## Definiții de metode abstracte.

Observați că este declarat doar antetul. Aceste metode trebuie suprascrisse (overridden) în toate clasele concrete.



# Exemplu: subclasa Circle

```
/**  
 * Concrete class Circle - inherits from Shape  
 */  
  
public class Circle extends Shape {  
    private double r; // radius of circle  
    // Constructor  
    public Circle(double r) {  
        super();  
        this.r = r;  
    }  
    // calculate area  
    public double calculateArea()  
        return Math.PI * r * r;  
    }  
    // calculate perimeter  
    public double calculatePerimeter()  
        return 2.0 * Math.PI * r ;  
    }  
    protected void finalize() throws Throwable  
        super.finalize();  
    }  
}
```

## Clasă concretă.

Clasa *nu trebuie* să conțină sau să moștenească metode abstracte. Metodele abstracte moștenite trebuie suprascrise.

## Definiții de metode concrete.

Observați că aici este declarat corpul metodei.



# Exemplu: subclasa Triangle

```
/**  
 * Concrete class Triangle - inherits from Shape  
 */  
  
public class Triangle extends Shape {  
    private double s; // side of Triangle  
    // Constructor  
    public Triangle(double s) {  
        super();  
        this.s = s;  
    }  
    // calculate area  
    public double calculateArea() {  
        return ( Math.sqrt(3.) / 4 * s * s );  
    }  
    // calculate perimeter  
    public double calculatePerimeter() {  
        return 3.0 * s ;  
    }  
    protected void finalize() throws Throwable {  
        super.finalize();  
    }  
}
```

**Clasă concretă.** Clasa *nu trebuie* să conțină sau să moștenească metode abstracte. Metodele abstracte moștenite trebuie suprascrise.

**Definiții de metode concrete.** Observați că corpurile metodelor sunt diferite de cele din Circle, dar semnăturile metodelor sunt *identice*.

Alte subclase ale lui Shape vor suprascrie și ele metodele abstracte calculateArea și calculatePerimeter



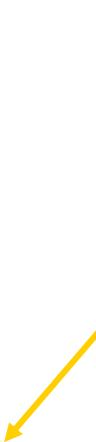
# Exemplu: clasa TestShape

```
/**  
 * Write a description of class TestShape here.  
 *  
 * @author (your name)  
 * @version (a version number or a date)  
 */  
  
public class TestShape  
{  
    public static void main(String[] args)  
    {  
        // Create an array of Shapes  
        Shape s[] = new Shape[2];  
        // create objects  
        s[0] = new Circle(2);  
        s[1] = new Triangle(2);  
        // Print out the number of Shapes  
        System.out.println(s[0].getCount() + " shapes created");  
        for (int i = 0; i < s.length; i++ ) {  
            System.out.print(s[i].toString() + " ");  
            System.out.print("Area = " + s[i].calculateArea());  
            System.out.println(" Perimeter = " + s[i].calculatePerimeter())  
        }  
    }  
}
```

**Creează obiecte ale subclaselor folosind referințe la superclasă.**



**Apelază metodele calculateArea și calculatePerimeter.**  
Este apelată automat versiunea corespunzătoare a fiecărei metode pentru fiecare obiect.



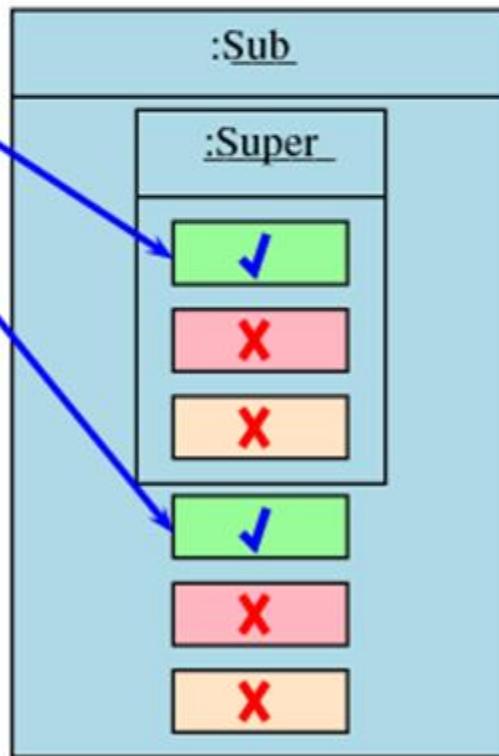
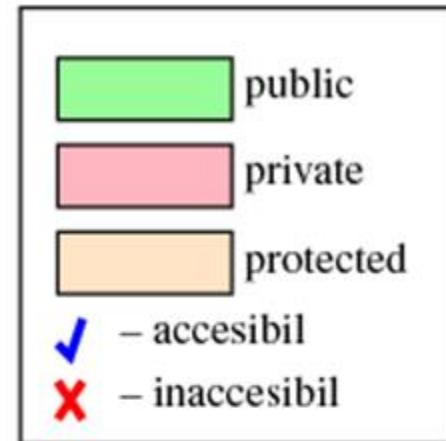
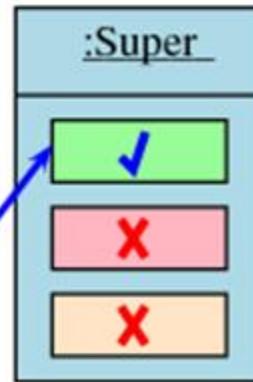


# Variabile instanță

- Ca şablon general, subclasele
  - Moştenesc capabilităatile **public** (metode)
  - Moştenesc proprietăatile **private** (variabile instanță) dar nu au acces la ele
  - Moştenesc variabilele **protected** și le pot accesa
- O variabilă declarată **protected** de o superclasă devine *parte a moştenirii*
  - Variabila devine disponibilă pentru subclase, care o pot accesa *ca și cum ar fi proprie*
  - Spre deosebire de aceasta, dacă o variabilă instanță este declarată **private** într-o superclasă, subclasele nu vor avea acces la ea
    - Superclasa poate totuși oferi acces protejat la variabilele instanță private via metode *accesoare* și *mutatoare*



Accesibilitatea din metodele Clientului



Doar membrii declarați public  
– definiți în cadrul clasei și  
cei moșteniți – sunt vizibili  
din exterior; celelalte  
elemente sunt ascunse  
vederii din exterior.



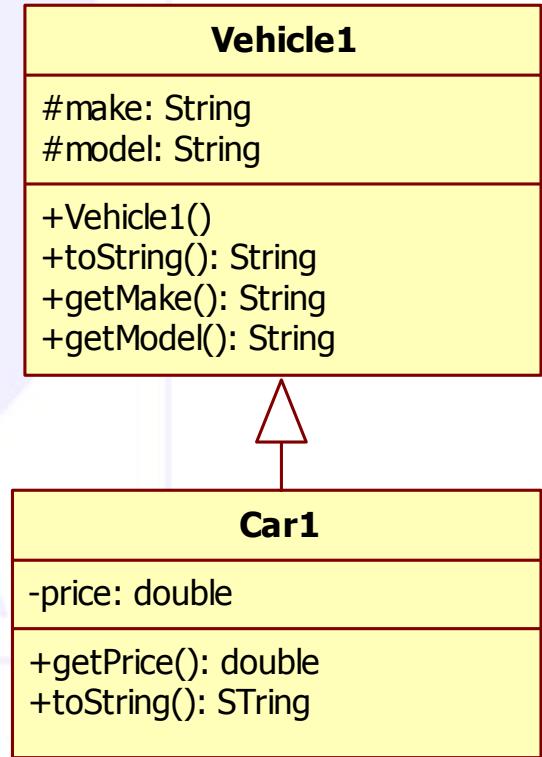
# Variabile instanță **protected** față de variabile instanță **private**

- Cum putem decide între **private** și **protected**?
  - folosiți **private** dacă doriți ca o variabilă instanță să fie *încapsulată* de către superclasă
    - d.e., ușile, ferestrele, bujiile unei mașini
  - folosiți **protected** dacă doriți ca variabila instanță să fie accesibilă subclaszelor pentru a o modifica (și nu doriți să faceți variabila mai general accesibilă prin metode accesoare/mutatoare)
    - d.e., motorul unei mașini



# protected, Exemplu

```
public class Vehicle1 {  
    protected String make;  
    protected String model;  
    public Vehicle1() { make = ""; model = ""; }  
    public String toString() {  
        return "Make: " + make + " Model: " + model;  
    }  
    public String getMake() { return make; }  
    public String getModel() { return model; }  
}  
  
public class Car1 extends Vehicle1 {  
    private double price;  
    public Car1() { price = 0.0; }  
    public String toString() {  
        return "Make: " + make + " Model: " + model  
            + " Price: " + price;  
    }  
    public double getPrice() { return price; }  
}
```





# Suprascrierea unei definiții de metodă

- Deși o clasă derivată moștenește metode din clasa de bază, ea poate să le modifice – să le *suprascrie* dacă este necesar
  - Pentru a suprascrie o definiție de metodă, se pune pur și simplu o definiție nouă în definiția clasei, exact ca pentru orice altă metodă adăugată clasei deriveate
- De obicei, tipul returnat nu poate fi schimbat la suprascrierea unei metode
- Totuși, dacă tipul este un *tip clasă*, atunci tipul returnat poate fi schimbat la acela al oricărei *clase descendente* al tipului returnat
- Acest lucru se cunoaște sub numele de ***tip returnat covariant***
  - *Tipurile returnate covariant* sunt introduse în Java 5.0; ele nu sunt permise în versiuni anterioare de Java



# Tipul returnat covariant

- Fiind dată următoarea clasă de bază:

```
public class BaseClass
{
    ...
    public BankAccount getAccount(int someKey)
    ...
}
```

- Este permisă următoarea modificare a tipului returnat în Java 5.0:

```
public class DerivedClass extends BaseClass
{
    ...
    public SavingsAccount getAccount(int someKey)
    ...
}
```

Computer Science



# Schimbarea permisiunii de acces a unei metode suprascrise

- Permisionea de acces a unei metode suprascrise poate fi schimbată *de la private* în *clasa de bază* la *public* (sau alt *acces mai permisiv*) în *clasa derivată*
- Totuși, permisiunea de acces a unei metode suprascrise *nu poate fi modificată* de la public în clasa de bază *la o permisiune de acces mai restrictivă* în clasa derivată
  - Adică, putem relaxa permisiunile de acces într-o clasă derivată, nu o putem restrânge

Computer Science



# Schimbarea permisiunii de acces a unei metode suprascrise

- Fiind dat următorul antet de metodă într-o clasă de bază:  
**private void doSomething()**
- Următorul antet de metodă este valid într-o clasă derivată:  
**public void doSomething()**
- Invers (din public în privat) nu se poate
- Fiind dat următorul antet de metodă într-o clasă de bază:  
**public void doSomething()**
- Antetul de metodă următor *nu* este valid într-o clasă derivată:  
**private void doSomething()**



# Capcană: Suprascriere față de supraîncărcare

- Nu confundați suprascrierea (*overriding*) unei metode într-o clasă derivată cu supraîncărcarea (*overloading*) numelui unei metode
  - Când o metodă este *suprascrisă*, noua definiție de metodă dată în clasa derivată are *exact același număr și tipuri de parametri ca în clasa de bază*
  - Când o metodă dintr-o clasă derivată are o *semnătură diferită* în comparație cu metoda din clasa de bază, atunci avem de-a face cu *supraîncărcarea*
  - Observați că atunci când *clasa derivată suprascrie* metoda originală, *ea totuși moștenește și metoda originală* din clasa de bază



# Modifierul **final**

- Dacă se pune modifierul **final** în fața definiției unei *metode*, atunci metoda respectivă *nu poate fi suprascrisă* într-o clasă derivată
- Dacă modifierul **final** este pus în fața definiției unei *clase*, atunci clasa respectivă *nu mai poate fi folosită pe post de clasă de bază* pentru a deriva alte clase

Computer Science



# Constructorul super

- O clasă derivată folosește un constructor al clasei de bază pentru a inițializa toate datele moștenite din clasa de bază
  - Pentru a invoca un constructor al clasei de bază, se folosește o sintaxă specială:

```
public DerivedClass(int p1, int p2, double p3)
{
    super(p1, p2);
    instanceVariable = p3;
}
```

- În exemplul de mai sus, `super(p1, p2);` este un apel al constructorului clasei de bază

Computer Science



# Accesul la o metodă redefinită din clasa de bază

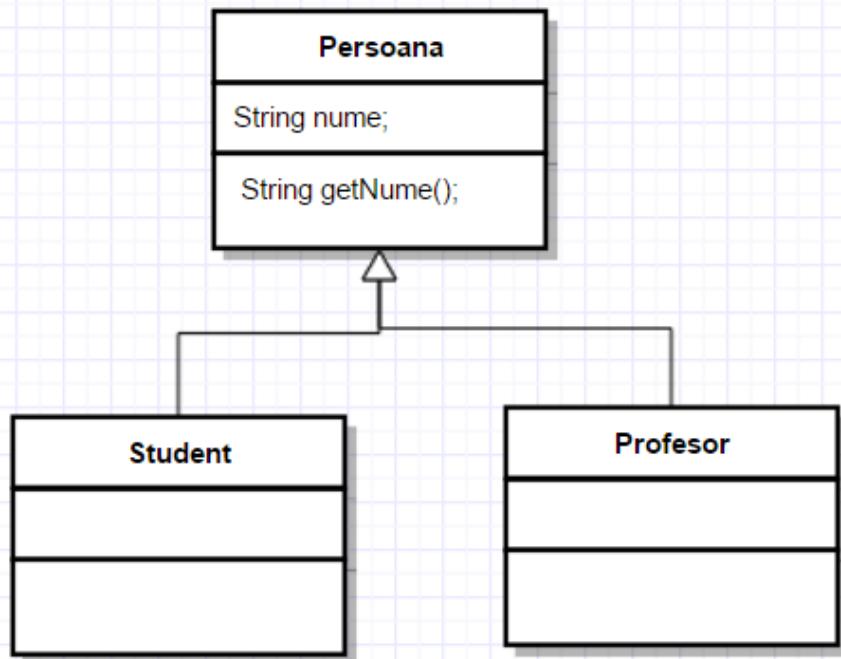
- În definiția unei metode dintr-o clasă derivată, versiunea suprascrisă a unei metode a clasei de bază poate totuși fi invocată
  - Pur și simplu prefixați numele metodei cu **super** și un punct

```
public String toString()
{
    return (super.toString() + "$" + interestRate);
}
```
- Cu toate acestea, la folosirea unui obiect al clasei derivate în afara definiției clasei, nu există nici o cale de invocare a versiunii unei metode suprascrise din clasa sa de bază



# Construirea obiectelor în Java

Exemplu de cod:



```
public class Persoana{
    private String nume;
    public String getNume() {
        return nume;
    }
}

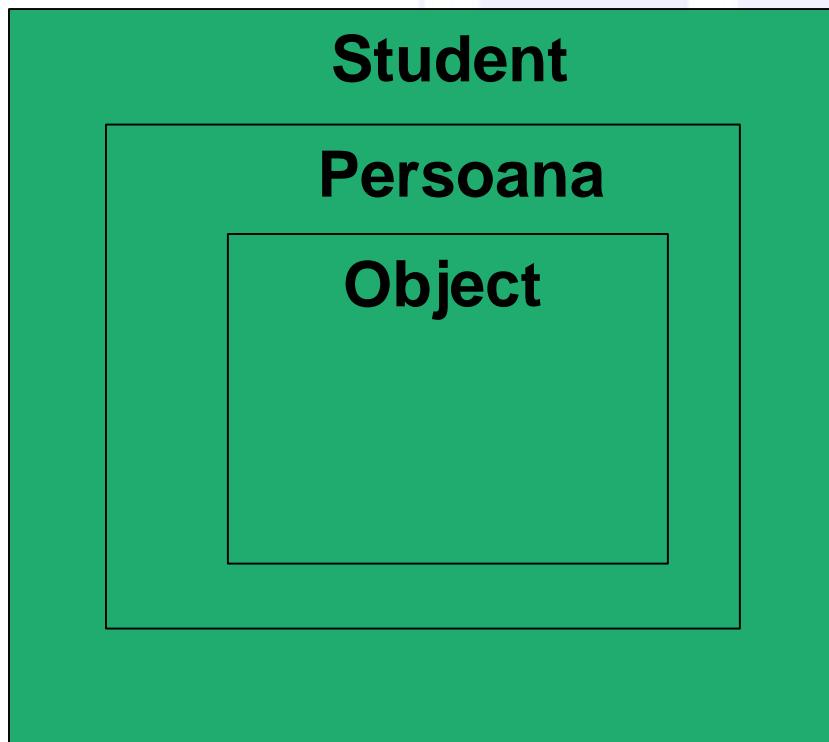
class Student extends Persoana{
}

class Profesor extends Persoana{
}
```



# Construirea obiectelor în Java

```
Student s = new Student();
```



Alocarea spațiului în memorie se face astfel:

- Se alocă spațiu pentru atributele din clasa **Object** (**atenție, clasa Object este moștenită implicit!**)
- Se alocă spațiu pentru atributele din clasa **Persoana**
- Se alocă spațiu pentru atributele din clasa **Student**



# Construirea obiectelor în Java

- Cum interpretează compilatorul Java codul din acest exemplu?
  - Regula **1**: dacă o clasă nu extinde o altă clasă, atunci compilatorul inserează implicit: **extends Object**

```
public class Persoana{  
    private String nume;  
    //...  
}
```

```
public class Persoana extends Object{  
    private String nume;  
    //...  
}
```



# Construirea obiectelor în Java

- Cum interpretează compilatorul Java codul din acest exemplu?
  - Regula 2: dacă într-o clasă nu este definit nici un constructor, compilatorul creează implicit constructorul fără parametri

```
public class Persoana{  
    private String nume;  
    //...  
}
```

```
public class Persoana extends Object{  
    private String nume;  
    Persoana(){  
    }  
    //...  
}
```



# Construirea obiectelor în Java

- Cum interpretează compilatorul Java codul din acest exemplu?
  - Regula 3: prima linie din interiorul constructorului trebuie să fie
    - fie apelul unui alt constructor: **this(<params>)**
    - fie apelul unui constructor din superclasa: **super(<params>)**
- **Altfel, compilatorul apelează implicit constructorul superclasei fără parametri super()**

```
public class Persoana{  
    private String nume;  
    //...  
}
```

```
public class Persoana extends Object{  
    private String nume;  
    Persoana(){  
        super();  
    }  
}
```



# Un obiect al unei clase derivate are mai mult de un tip

- Un obiect al unei clase derivate are tipul clasei derivate și are și tipul clasei de bază
- Mai general, un obiect al unei *clase derivate* are *tipul fiecărui dintre clasele din ascendența sa*
  - De aceea, un obiect dintr-o clasă derivată poate fi asignat unei variabile de tipul oricărui părinte/strămoș al său
  - Observați, totuși, că relația nu merge și invers!

Computer Science



# Polimorfism

- Poli = mai multe
- Morphos = forme
- Polimorfismul se referă la această proprietate a obiectelor de a avea mai multe forme
- Spre exemplu, un obiect de tip Persoana poate referi spre un obiect de tip Student:  
**Persoana p= new Student("Ana", 2854);**

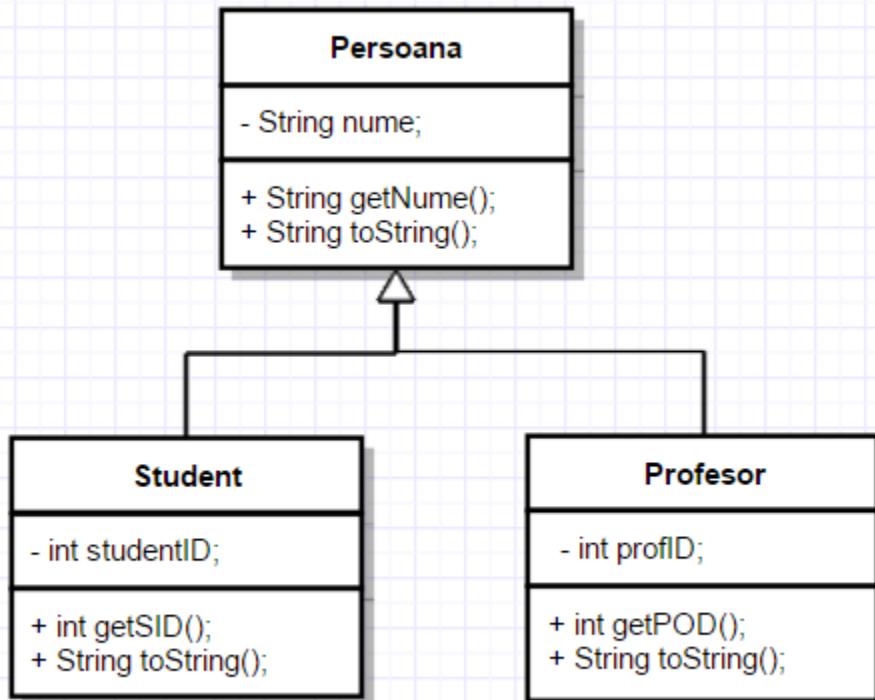


# Polimorfism

Dându-se diagrama de clase alăturată,  
ce va afișa următorul cod?

```
Persoana p[] = new Persoana[3];
p[0] = new Persoana("Ion");
p[1] = new Student("Ana", 1234);
p[2] = new Profesor("Mara", 8);
for(int i = 0; i < p.length; i++) {
    System.out.println( p[i] );
}
```

Rezultate afișate:  
**Ion**  
**1234: Ana**  
**8: Mara**





# Polimorfism

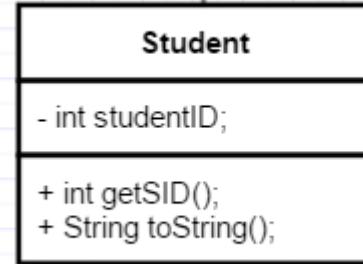
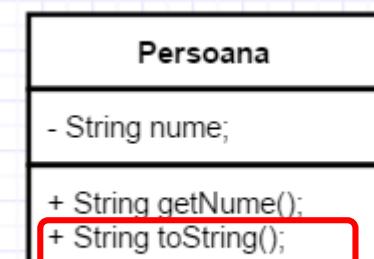
Decizii luate la compilare vs. în timpul execuției

## ■ Reguli pentru compilare

- Compilatorul cunoaște doar tipul referință al obiectului
- Caută în clasa tipului referință dacă există metoda care se dorește a fi apelată
- și returnează antetul metodei (semnătura)

Semnătura metodei:  
**String toString();**

```
Persoana p = new Student("Ana", 2854);  
p.toString();
```





# Polimorfism

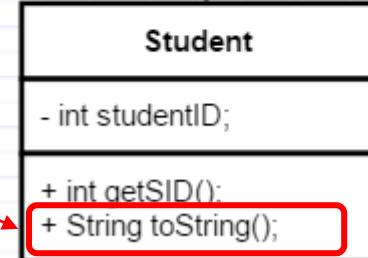
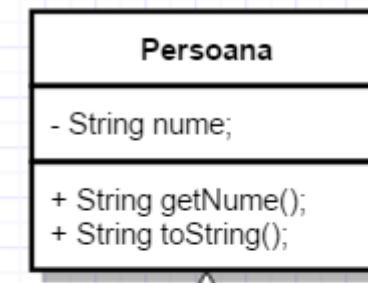
Decizii luate la compilare vs. în timpul execuției

## ■ Reguli pentru execuție

- Se va urma tipul obiectului creat efectiv în momentul execuției
- Semnătura returnată în momentul compilării trebuie să se potrivească cu metoda din clasa actuală a obiectului
  - În cazul în care metoda nu este gasită în clasa actuală, se caută mai sus în ierarhia de clase

```
Persoana p = new Student("Ana", 2854);  
p.toString();
```

Metoda care se execută:





# Polimorfism

Decizii luate la compilare vs. în timpul execuției

- Ce se întâmplă la execuția următoarelor variante de cod?
  - Persoana p = new Student("Ana",1234); p.getSID();

**R: Eroare de compilare**

**Soluție: ((Student) p).getSID();**

- pentru a evita erorile la execuție, folosiți:

```
if( p instanceof Student ) {  
    // se execută doar dacă p "este-un" Student la execuție  
    ( (Student)s ).getSID();  
}
```

- Student s = new Persoana("Ion");

**R: Eroare de compilare**

**Solutie: - nu există**



# Polimorfism. Exemplu 1

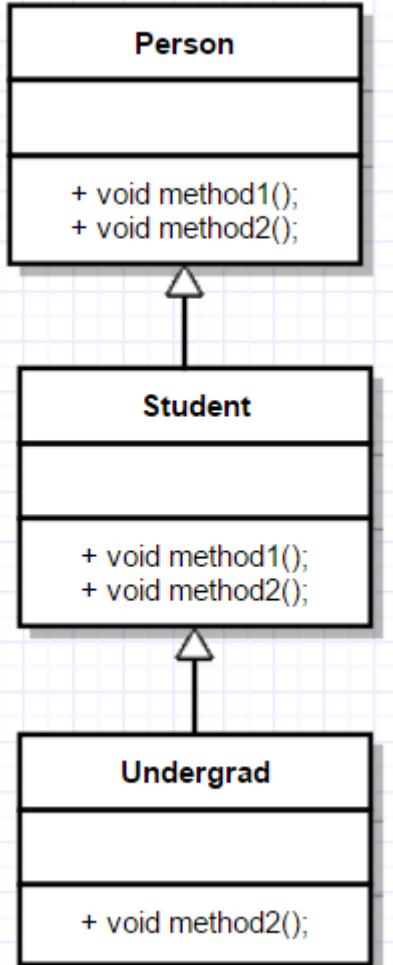
```
public class Persoana {  
    private String name;  
    public Person(String name) {  
        this.name = name;  
    }  
    public boolean isAsleep(int hr) {  
        return 22 < hr || 7 > hr;  
    }  
    public String toString() {  
        return name;  
    }  
    public void status(int hr) {  
        if (this.isAsleep(hr))  
            System.out.println("Now  
offline: " + this);  
        else  
            System.out.println("Now  
online: " + this);  
    }  
}
```

```
public class Student extends  
Persoana {  
    public Student(String name) {  
        super(name);  
    }  
    public boolean isAsleep(int hr) {  
        //suprascriere  
        return 2 < hr && 8 > hr;  
    }  
    public static void main(String[]  
                           args) {  
        Persoana p;  
        p = new Student("Ana");  
        p.status(1);  
    }  
}
```

Rezultate afișate:  
Now online: Ana



# Polimorfism. Exemplu 2



```
public class Person {
    public void method1() {
        System.out.print("Person 1 ");
    }
    public void method2() {
        System.out.print("Person 2 ");
    }
}

class Student extends Person {
    public void method1() {
        System.out.print("Student 1 ");
        super.method1();
        method2();
    }
    public void method2() {
        System.out.print("Student 2 ");
    }
}

class Undergrad extends Student {
    public void method2() {
        System.out.print("Undergrad 2 ");
    }
}
```



```
public class Person {  
    public void method1() {  
        System.out.println("Person 1 ");  
    }  
    public void method2() {  
        System.out.println("Person 2 ");  
    } }  
  
class Student extends Person {  
    public void method1() {  
        System.out.println("Student 1 ");  
        super.method1();  
        method2(); //this.method2();  
    }  
    public void method2() {  
        System.out.println("Student 2 ");  
    } }  
  
class Undergrad extends Student {  
    public void method2() {  
        System.out.println("Undergrad 2 ");  
    } }
```

- Ce se va afișa la executarea următoarelor linii de cod?

```
Person p = new Undergrad();  
p.method1();
```

1

#### Rezultate afișate:

Student 1  
Person 1  
Undergrad 2



# Polimorfism. Exemplu 2

## ■ Discuții:

1. Se execută întâi method1() din clasa Student, deoarece în clasa Undergrad nu există o metodă cu acestă semnătură, astfel se execută prima metodă gasită mergând înapoi sus în ierarhia de clase. Se afișează "Student 1"
2. Apoi se apelează method1() din clasa Person (indicată de apelativul *super*, care în momentul compilării stabilește că apelul trebuie făcut către method1() din clasa Person). Se afișează "Person 1"
3. Se apelează method2() din clasa Undergrad, deoarece compilatorul interpretează apelul "method2();" ca "**this.method2()**", unde this se referă la obiectul din care se face apelul, și anume obiectul concret creat în momentul execuției, care este de tip Undergrad



# Polimorfism

- Reguli în ceea ce privește apelul metodelor folosind operatorii **this** și **super**:
  - Când apelăm o metodă cu **super** (ex: super.method1()), legarea se face la compilare
    - Atunci se verifică care e clasa părinte
  - Când apelăm o metodă cu **this** (ex. this.method2(), sau pur și simplu method2()), legarea se face în momentul execuției, în funcție de tipul concret al obiectului creat
    - Aceasta mai poartă numele de **legare dinamică**

Computer Science



# Polimorfism. Legarea dinamică

- Apare atunci când decizia privind metoda de executat nu se poate lua decât la execuția programului
- Este nevoie de ea atunci când
  - Variabila este declarată ca având tipul superclasei și
  - Există mai mult de o metodă polimorfică care se poate executa între tipul variabilei și subclasele sale



# Cum se decide care este metoda de executat?

1. Dacă există o metodă concretă în clasa curentă, se execută aceea
2. În caz contrar, se verifică în superclasa directă dacă există acolo o metodă; dacă da, se execută
3. Se repetă pasul 2, verificând în sus pe ierarhie până când se găsește o metodă concretă și se execută
4. Dacă nu s-a găsit nici o metodă, atunci Java semnalează o eroare de compilare



# Polimorfism

- O variabilă polimorfică poate părea a-și schimba tipul prin legare dinamică
- Compilatorul înțelege întotdeauna tipul unei variabile potrivit declarației
- Compilatorul permite o anume flexibilitate prin modul de conformare la tip
- La execuție, comportamentul unui apel de metodă depinde de *tipul de obiect*, nu de *variabilă*
- Exemplu:

```
Person p;  
p = new Student();  
p = new Undergrad();  
p.method1();
```



# De ce este util polimorfismul?

- Polimorfismul permite unei superclase să rețină ceea ce este comun, lăsând specificitatea să fie tratată de subclase

Să presupunem că AView include o metodă **calcArea**, ca mai sus

Atunci ARectangle trebuie scris ca ...

iar AOval trebuie scris ca ...

Considerați acum

```
public class AView {  
    ...  
    public double calcArea() {  
        return 0.0;  
    }  
}
```

```
public class ARectangle extends AView {  
    ...  
    public double calcArea() {  
        return getWidth() * getHeight();  
    }  
}
```

```
public class AOval extends AView {  
    ...  
    public double calcArea() {  
        return getWidth()/2. * getHeight()/2. * Math.PI;  
    }  
}
```

```
public double coverageCost(AView v, double costPerSqUnit) {  
    return v.calcArea() * costPerSqUnit;  
}
```



# Interfețe, clase abstracte și clase concrete

- O **interfață**
  - se folosește pentru a specifica funcționalitatea cerută de un client
- O **clasă abstractă**
  - oferă o bază pe care să se construiască clase concrete
- O **clasă concretă**
  - completează implementarea efectivă a metodelor abstracte care au fost specificate de o interfață sau printr-o clasă abstractă
  - furnizează obiecte la momentul execuției
  - nu este, în general, potrivită ca bază pentru extindere



# Folosirea claselor abstracte

- O clasă abstractă contribuie la implementarea subclaserelor sale concrete
- Este folosită pentru a exploata *polimorfismul*
  - Pentru funcționalitatea specificată în clasa părinte se pot da implementări corespunzătoare fiecărei subclase concrete
- Clasele abstracte trebuie să fie stabile
  - Orice schimbare într-o clasă abstractă se propagă la subclase și la clientii lor
- O clasă concretă poate extinde doar o singură clasă (abstractă sau concretă)

Computer Science



# Folosirea interfețelor

- Interfețele sunt abstracte prin definiție
  - Separă implementarea unui obiect de specificarea sa
  - Nu fixează nici un aspect al unei implementări
- O clasă poate implementa mai mult de o interfață
- Interfețele permit o folosire mai generalizată a polimorfismului; instanțe din clase relativ neînrudite pot fi tratate ca identice într-un scop anume
- În programe, folosiți
  - *interfețe pentru a partaja comportament comun*
  - *moștenirea pentru a partaja cod comun*

Computer Science



# Programare orientată pe obiecte

1. Clasele Object și Class
2. Interfețe Java

Computer Science



# Metode din clasa Object

- *Object* definește versiuni implicite ale următoarelor metode:
  - **toString()** – returnează un **String** (reprezentare "citibilă" a obiectului)
  - **equals(Object obj)** – returnează egalitatea referințelor; trebuie să fie suprascrisă pentru a realiza egalitatea de conținut în subclase
  - **hashCode()** – returnează valoarea codului de dispersie pentru obiect; valorile sunt diferite pentru obiecte diferite
  - **getClass()** – returnează un obiect de tipul Class; există un obiect de tipul Class pentru fiecare clasă dintr-o aplicație
  - **notify()**, **notifyAll()**, **wait()**, **wait(long timeout)**, **wait(long timeout, int nanos)** – folosite la *multithreading*
  - **clone()** – creează și întoarce o copie a acestui obiect; "copie" poate depinde de clasa obiectului
  - **finalize()** – destinat a efectua acțiuni de "curățare" înainte ca obiectul să fie irevocabil abandonat



# Egalitatea

- Există două feluri diferite de egalitate:
  - *Egalitatea de identitate* care înseamnă că două expresii au aceeași identitate (adică reprezintă același obiect)
    - Simbolul == testează egalitatea de identitate atunci când este aplicat datelor referință
  - *Egalitatea de conținut* care înseamnă că două expresii reprezintă obiecte cu aceeași valoare/conținut
    - Metoda *equals* din **Object** returnează **true** dacă și numai dacă este invocată cu două referințe identice; metoda poate fi suprascrisă în orice subclasă pentru a verifica egalitatea de conținut
- Exemplu:

```
AOval ov1, ov2;  
ov1 = new AOval(0, 0, 100, 100);  
ov2 = new AOval(0, 0, 100, 100);  
if (ov1 == ov2){ System.out.println("Egalitate de identitate");}  
if (ov1.equals(ov2)){ System.out.println("Egalitate de continut");}
```



# Clasa **Class**

- Clasa **Class** este definită astfel:  

```
public final class Class extends Object  
    implements Serializable, ...
```
- Instanțele clasei **Class** reprezintă clase și interfețe dintr-o aplicație Java în curs de execuție
- Un obiect de tipul **Class** conține informații despre clasa a cărei instanță este obiectul care apelează
- Nu are constructor propriu
- Obiectele **Class** sunt construite la execuție de către JVM
- Există două moduri pentru a construi obiecte de acest tip:
  - **getClass ()** din clasa **Object**
  - **forName ()** din clasa **Class** (metodă statică)



# Clasa Class

- Metode:
  - **public String getName()**
    - returnează un **String** care reprezintă numele entității reprezentate de obiectul **Class this**
    - entitatea poate fi: clasă, interfață, tablou, tip primitiv, void
  - **public static Class forName(String className) throws ClassNotFoundException**
    - returnează un obiect de tipul **Class** care conține informații despre clasa obiectului
  - **public Class[] getClasses()**
    - returnează un tablou de obiecte de tip **Class**;
    - toate clasele și interfețele, membri publici ai clasei sunt reprezentate de acest obiect **Class**



# Clasa Class

## ■ Metode (continuare)

### ■ **Field[] getFields**

- returnează un tablou care conține obiecte **Field** care reflectă toate câmpurile accesibile public ale clasei sau interfeței reprezentate de acest obiect **Class**

### ■ **Method[] getMethods()**

- returnează un tablou care conține obiecte **Method** care reflectă toate metodele publice *membre* ale clasei sau interfeței reprezentate de acest obiect **Class**, inclusiv cele declarate de clasă sau interfață și cele moștenite din superclase și superinterfețe

### ■ **Constructor[] getConstructors()**

- returnează un tablou care conține obiecte **Constructor** care reflectă toți constructorii publici ai clasei sau interfeței reprezentate de acest obiect **Class**



# Operatorul instanceof

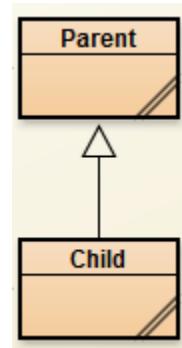
- Operatorul **instanceof** verifică dacă un obiect este de tipul dat ca al doilea argument al său

**Obiect instanceof NumeClasa**

- Va returna **true** dacă **Obiect** este de tipul **NumeClasa**; altfel va returna **false**
- Observați că aceasta înseamnă că va returna **true** dacă **Obiect** are tipul *oricărei clase care este descendenta* a lui **NumeClasa**

Exemplu:

```
Child c = new Child("Ana");  
System.out.print(c instanceof Child); //true  
System.out.print(c instanceof Parent); //true
```





# Metoda `getClass()`

- Fiecare obiect moștenește aceeași metodă `getClass()` din clasa `Object`
  - Această metodă este marcată `final`, deci nu poate fi suprascrisă
- O invocare a lui `getClass()` pe un obiect returnează o reprezentare *numai* pentru clasa care a fost folosită cu operatorul `new` pentru a crea obiectul
  - Ex. `Parent p = new Child("Bubu");  
System.out.print(p.getClass()); // Child`
  - Rezultatele a oricare două asemenea invocări pot fi comparate cu `==` sau `!=` pentru a determina dacă ele reprezintă sau nu aceeași clasă  
`(obiect1.getClass() == obiect2.getClass())`



# instanceof și getClass ()

- Atât operatorul `instanceof` cât și metoda `getClass()` se pot folosi pentru a verifica clasa unui obiect
- Totuși, metoda `getClass()` este mai exactă
  - Operatorul `instanceof` doar testează clasa unui obiect
  - Metoda `getClass()` folositoare testă dacă două obiecte *au fost instanțiate* din aceeași clasă



# Exemple

- Afișarea numelui unei clase folosind un obiect de tip **Class**

```
void printClassName(Object obj) {  
    System.out.println(obj + " este de clasa " +  
        obj.getClass().getName());  
}
```

- Alte exemple

```
Circle c = new Circle(5);  
printClassName(c); //Cercul cu raza 5 este de clasa Circle  
Class c1 = c.getClass();  
System.out.println(c1.getName()); // "Circle"  
Triangle t = new Triangle(7);  
printClassName(t); //Triunghiul cu laturile 7 este de clasa Triangle  
try {  
    Class c2 = Class.forName("Triangle");  
    System.out.println(c2.getName()); // "Triangle"  
}  
catch (ClassNotFoundException e) {  
    System.err.println("Nu exista clasa \"Triangle\" " +  
        e.getMessage());  
}
```



# Specificațiile și Java

- Un program este asamblat dintr-o colecție de clase care trebuie să "lucreze împreună" sau să "se potrivească una cu alta"
- Limbajul și compilatorul Java ne pot ajuta să:
  - Scriem specificații de clase și să
  - Verificăm că o clasă satisfac corect (implementează) specificațiile sale
- Există mai multe construcții în Java:
  - Construcția **interface** – ne permite să codificăm în Java informația pe care o specificăm, d.e. într-o diagramă de clasă
  - Construcția **extends** – ne permite să codificăm o clasă prin adăugarea de metode la o clasa existentă
  - Construcția **abstract class** – ne permite să codificăm o clasă incompletă care poate fi încheiată (terminată) printr-o altă clasă



# Un exemplu

- Două persoane lucrează la același proiect, în același timp:
  - o persoană modelează un cont bancar
  - o alta scrie o clasă pentru plăți lunare din cont
- Pentru a realiza acest lucru, cei doi trebuie să se înțeleagă cu privire la *interfață*, de exemplu:

```
/** SpecificatieContBancar specifică modul de comportare al contului bancar. */
public interface SpecificatieContBancar
{
    /** depune adaugă bani în cont
     * @param suma - suma de bani de depus, un întreg nenegativ */
    public void depune(int suma);

    /** retrage scoate bani din cont dacă se poate
     * @param suma - suma de retras, un întreg nenegativ
     * @return true, dacă retragerea a avut succes;
     *         false, în caz contrar. */
    public boolean retrage(int suma);
}
```



# Ce este o interfață?

- *Interfața* spune că, indiferent de clasa scrisă pentru a implementa o **SpecificatieContBancar**, clasa respectivă trebuie să conțină două metode, **dăpune** și **retrage**, care să se comporte aşa cum s-a precizat
- În general, o *interfață* este un dispozitiv sau un sistem pe care entități ne-înrudite îl folosesc pentru a interacționa
- Exemple:
  - O telecomandă reprezintă interfață dintre persoană și televizor,
  - Limba română este o interfață între două persoane care o vorbesc
  - Protocolul de comportament din armată reprezintă interfață dintre indivizi de diferite grade

Computer Science



# Ce este o interfață?

- În Java: o *interfață* este un tip, aşa cum și o clasă este un tip
  - Asemănător unei clase, o interfață *definește metode*
  - Spre deosebire de o clasă, o interfață *nu implementează niciodată metode (valabil în Java 7 sau versiuni mai vechi); totuși începând cu Java 8 sunt permise și implementări implicitе (default) de metode*
  - Clasele care implementează interfața implementează metodele definite de interfață
  - O clasă poate implementa mai multe interfețe
- O interfață se folosește pentru a defini un *protocol de comportament* care poate fi implementat de către orice clasă de oriunde din ierarhia de clase



# Definiție. Utilitate

- Definiție: *o interfață este o colecție de constante și definiții de metode, colecție care are un nume*
- O interfață nu este o clasă, ci un set de *cerințe* pentru clasele care doresc să se conformeze interfeței
- Interfețele sunt folositoare pentru
  - *Reținerea asemănărilor* între *clase ne-înrudite* fără a forța o relație de clasă
  - *Declararea de metode* pe care una sau mai multe clase ar trebui să le implementeze
  - *Dezvăluirea interfeței de programare* a unui obiect fără a-i dezvălui clasa
  - *Modelarea moștenirii multiple*, care permite ca o clasă să aibă mai mult de o superclasă



# Definirea unei interfețe

- Definirea unei interfețe are două componente: declarația interfeței și corpul interfeței
  - *Declarația* interfeței definește diferențele atributelor interfeței, cum sunt numele și dacă ea extinde alte interfețe
  - *Corpul* interfeței conține declarațiile de constante și de metode pentru interfața respectivă



# Definirea unei interfețe

## ■ Sintaxa:

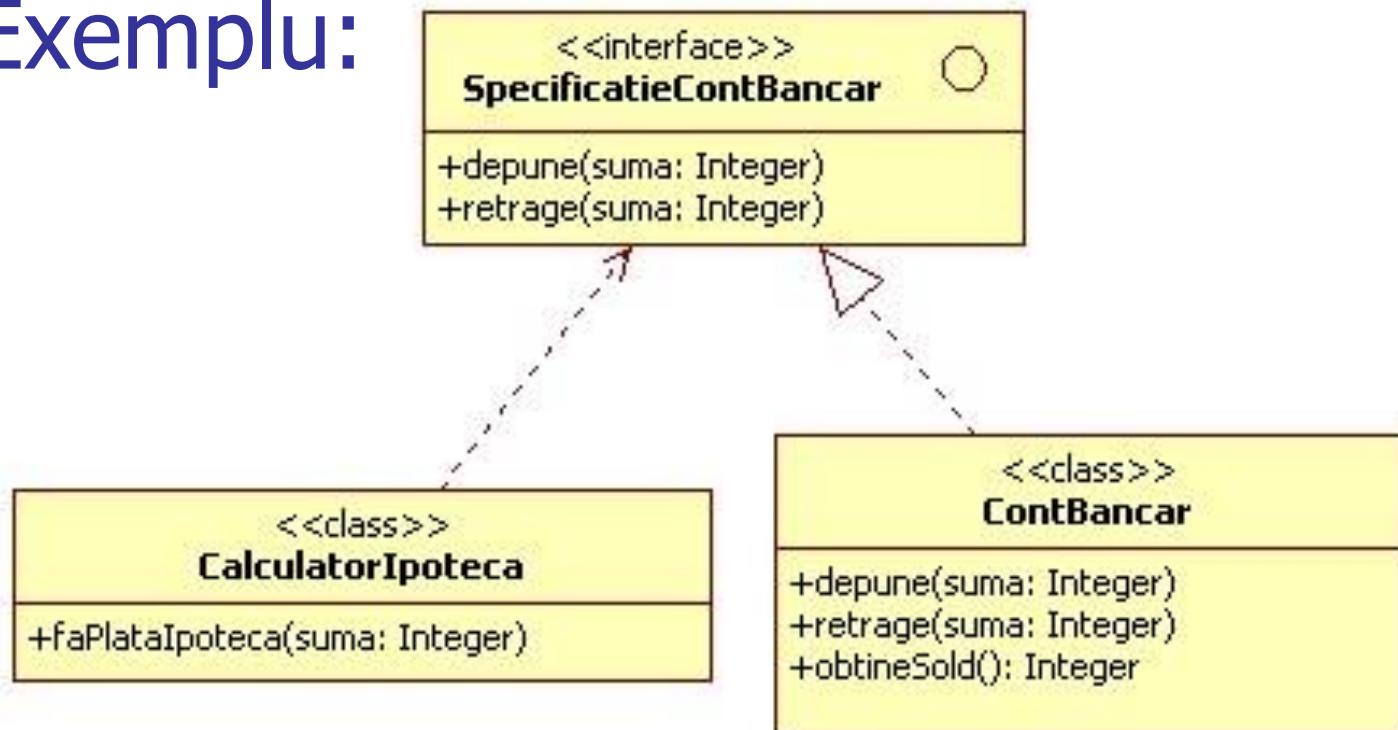
```
[modificator acces] interface NumeInterfata [extends AltaInterfata]{  
    /* zona declarare constante */  
    public static final int X = 10;  
  
    /*zona declarare metode*/  
    public void numeMetoda();  
}
```

## ■ Implementarea unei interfețe:

```
class NumeClasa implements NumeInterfata{  
    public void numeMetoda(){  
        // codul necesar pentru implementarea metodei  
    }  
}
```



# Exemplu:



- Pentru a conecta cele două clase – scrieți o metodă de lansare cam aşa:

```
ContBancar contulMeu = new contBancar();
CalculatorIpoteca calc = new CalculatorIpoteca(contulMeu);
...
calc.faPlataIpoteca(500);
```



# O clasă care implementează o interfață

```
/** ContBancar gestioneaza un singur cont bancar; cum este precizat
 * in antetul sau, el implementeaza SpecificatieContBancar: */
public class ContBancar implements SpecificatieContBancar{
    private int sold; // soldul contului

    /** Constructor ContBancar initializeaza contul */
    public ContBancar() { sold = 0; }

    // implementarea metodelor din interfata SpecificatieContBancar:
    public void depune(int suma) { sold = sold + suma; }
    public boolean retrage(int suma) {
        boolean rezultat = false;
        if ( suma <= sold ) {
            sold = sold - suma;
            rezultat = true;
        }
        return rezultat;
    }
    /** cerSold raporteaza soldul current; @return soldul */
    public int cerSold() { return sold; }
}
```



# O clasă care referă o interfață

```
/** CalculatorPlatiIpoteca face plati de ipoteca */
public class CalculatorIpoteca{
    private SpecificatieContBancar _contBancar; // pastreaza adresa

    // unui obiect care implementeaza SpecificatieContBancar
    /** Constructor CalculatorPlatiIpoteca initializeaza calculatorul.
     * @param cont - adresa contului bancar in/din care se fac
     * depuneri/retrageri */
    public CalculatorIpoteca(SpecificatieContBancar cont)
    {   _contBancar = cont; }

    /** faPlataIpoteca efectueaza o plata de ipoteca din contul bancar.
     * @param suma - suma de platit */
    public void faPlataIpoteca(int suma){
        boolean ok = _contBancar.retrage(suma);
        if ( ok )
        {   System.out.println("Plata efectuata: " + suma); }
        else { ... error ... }
    }
    ...
}
```



# Restrictii pentru interfețe

- Toate metodele unei interfețe trebuie să fie metode de instanță **abstract**; începând cu Java 8 *se permit și metode de instanță cu comportament implicit și metode statice*
- Toate variabilele definite într-o interfață trebuie să fie **static final**, adică *constante*
  - Valorile se pot stabili la compilare sau se pot calcula la încărcarea clasei
  - Variabilele pot fi de orice tip
- Nu sunt permise blocuri de inițializare statice
  - Fiecare inițializare trebuie să fie o linie pentru o variabilă
  - Începând cu Java 8, sunt permise metode statice pentru inițializare în interfață



# Instantierea

- Pentru a putea folosi metode ale instanței dintr-o interfață trebuie să existe un obiect asociat care implementează interfața
- Nu se poate instanta o interfață direct, dar se poate instanta o clasă care *implementează* interfața
  - Exemplu

```
SpecificatieContBancar contulMeu = new ContBancar();
```
- Referințele la un obiect se pot face via
  - numele clasei,
  - unul dintre numele superclaselor sale sau
  - unul dintre numele interfețelor sale



# Ce se pune într-o interfață

- Este considerat stil prost a scrie o interfață numai cu constante (static final)
  - De obicei acești calificatori sunt omiși:

```
interface MyConstants{  
    double PI = 3.141592;  
    double E = 1.7182818;  
}
```

Pot fi accesate fie ca **MyConstants.PI** sau doar PI de orice clasă care implementează interfața

- O interfață ar trebui să aibă cel puțin o metodă abstractă
- Dacă tot ce doriți este o colecție de constante, folosiți o clasă obișnuită cu **import static**

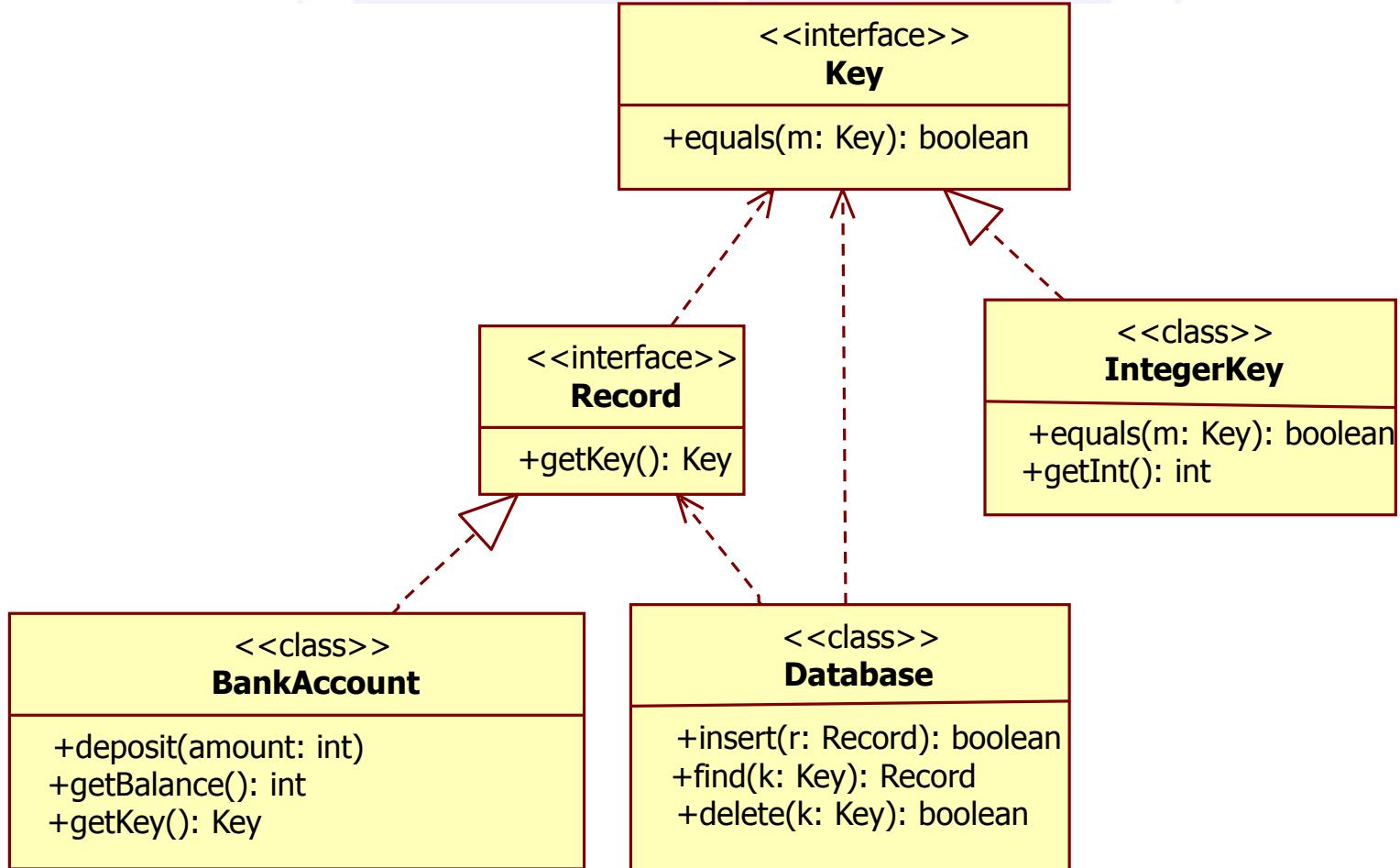


# Un alt exemplu. O "bază de date"

- Proiectați o clasă numită **Database**, care să păstreze o colecție de obiecte "înregistrare" (record), fiecare având o cheie (key) unică pentru identificare
- Comportamente esențiale – o specificație neformală:
  - **Database** păstrează o colecție de obiecte **Record**, unde fiecare **Record** are un obiect **Key**. Restul structurii oricărei **Record** nu are importanță și nu este cunoscut bazei de date
  - **Database** va avea metode **insert**, **find** și **delete**
  - Înregistrările **Record** indiferent de structura lor internă, vor avea o metodă **getKey** care returnează obiectul cheie (**Key**) al înregistrării (**Record**)
  - Obiectele **Key** vor avea o metodă **equals** care să compare două chei dacă sunt egale și să returneze true sau false



# Diagrama de clase cu interfețe





# Interfețe pentru Record și Key

```
/** Record este un element de date care poate fi stocat într-o bază de date */
public interface Record{
    /** getKey returnează cheia care identifică în mod unic înregistrarea
     * @return obiectul de tip Key din înregistrare */
    public Key getKey();
}

/** Key reprezintă o valoare pentru identificare, o "cheie" */
public interface Key{
    /** equals compara pe sine cu o altă cheie, m, ca să determine dacă sunt egale
     * @param m - celalaltă cheie
     * @return true, dacă aceasta cheie și m au aceeași valoare a cheii;
     *         returnează false, în caz contrar */
    public boolean equals(Key m);
}
```



# Superinterfețe

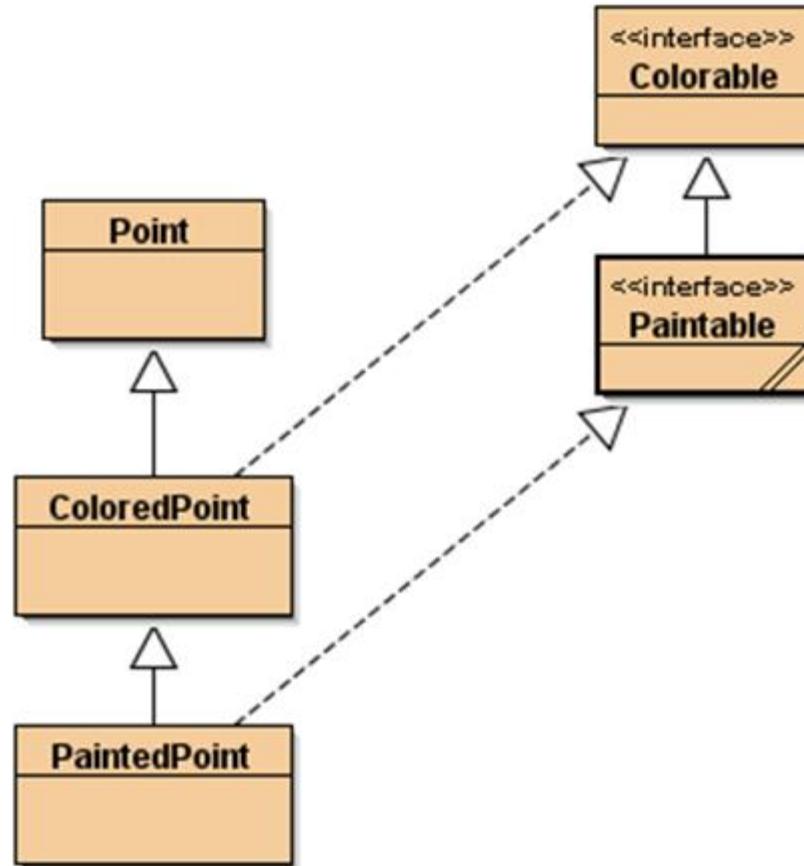
- Dacă este furnizată o clauză **extends**, atunci interfața în curs de declarare extinde fiecare dintre celelalte interfețe numite și din acest motiv moștenește metodele și constantele fiecărei dintre celelalte interfețe numite
- Aceste alte interfețe sunt numite *superinterfețe directe* ale interfeței în curs de declarare
- Orice clasă care implementează – **implements** – interfața declarată se consideră că *implementează toate interfețele pe care această interfață le extinde și care sunt accesibile clasei*

Computer Science



# Super/sub interfete. Exemplu

- Diagrama de clase





# Super/sub interfețe. Exemplu

```
public interface Colorable {  
    void setColor(int color);  
    int getColor();  
}  
  
public interface Paintable extends Colorable {  
    int MATTE = 0, GLOSSY = 1;  
    void setFinish(int finish);  
    int getFinish();  
}  
  
class Point {  
    int x, y;  
}  
  
class ColoredPoint extends Point implements  
    Colorable {  
    int color;  
    public void setColor(int color) {  
        this.color = color;  
    }  
    public int getColor() {  
        return color;  
    }  
}
```

```
class PaintedPoint extends ColoredPoint  
    implements Paintable {  
    int finish;  
    public void setFinish(int finish) {  
        this.finish = finish;  
    }  
    public int getFinish() {  
        return finish;  
    }  
}
```

- Interfața **Paintable** este o *subinterfață* a interfeței **Colorable**
- Interfața **Colorable** este o *superinterfață* a lui **Paintable**



# Interfețe predefinite. Exemplu

## Interfața Comparable

- Interfața Comparable este definită în pachetul `java.lang`, fiind automat disponibilă fiecărui program
- Nu are decât următoarea metodă care trebuie implementată:  
**`public int compareTo(Object other);`**
- Este responsabilitatea programatorului să urmeze semantica interfeței Comparable atunci când o implementează
- Metoda `compareTo` trebuie să returneze
  - Un număr negativ atunci când un obiect "este înainte de" parametrul `other`
  - Zero atunci când obiectul apelant "este egal cu" parametrul `other`
  - Un număr pozitiv dacă obiectul apelant "urmează după" parametrul `other`
- Semantica lui "`equals`" din metoda `compareTo` ar trebui să coincidă cu cea a metodei `equals` dacă se poate, dar aceasta nu este absolut necesar



# Interfață Comparable. Exemplu

```
import java.util.*;  
public class Person implements Comparable<Person>{  
    // instance variables - replace the example below with your own  
    private String nume;  
    private String prenume;  
    private int varsta;  
    //constructor  
    public Person(String n, String pr, int v) {  
        // initialise instance variables  
        nume = n;  
        prenume = pr;  
        varsta = v;  
    }  
    //@implements: compareTo(Object p) in Object  
    public int compareTo(Person p) {  
        return nume.compareTo(p.nume);  
    }  
    //@Overrides: toString() in Object  
    public String toString(){  
        return (nume+ " " + prenume + ", " + varsta + " ani");  
    }  
}
```



# Interfață Comparable. Exemplu

```
public class Test {  
  
    public static void afisare(Person[] v) {  
        for (int i = 0; i < v.length; i++)  
            System.out.println(v[i].toString());  
    }  
  
    public static void main() {  
        Person[] v = new Person[3];  
        v[0] = new Person("Pop", "Ioan", 20);  
        v[1] = new Person("Ana", "Maria", 15);  
        v[2] = new Person("Popescu", "Daria", 18);  
  
        System.out.println("Vectorul initial de persoane:");  
        afisare(v);  
        Arrays.sort(v);  
        System.out.println("\nVectorul sortat dupa nume:");  
        afisare(v);  
    }  
}
```

## Rezultatul afișat:

Vectorul initial de persoane:  
Pop Ioan, 20 ani  
Ana Maria, 15 ani  
Popescu Daria, 18 ani

Vectorul sortat dupa nume:  
Ana Maria, 15 ani  
Pop Ioan, 20 ani  
Popescu Daria, 18 ani



# Sumar interfețe: Declararea constantelor în interfețe

- Definiții de constante care nu specifică explicit modificatorii recomandați; totusi, toate exemplele sunt identice:

```
public static final int x = 1; // ceea ce se obtine implicit  
public int x = 1;  
int x = 1;  
static int x = 1;  
final int x = 1;  
public static int x = 1;  
public final int x = 1;  
static final int x = 1
```

- Oricare din aceste combinații este legală, dar implicit toate variabilele declarate în interfețe vor avea modificatorii

**public static final**



# Sumar interfețe: Implementarea interfețelor

- Interfețele sunt niște contracte care obligă o clasă să implementeze anumite funcționalități, fără ca să spună nimic despre cum trebuie implementate
- Interfețele pot fi implementate de orice clasă, de la orice nivel din arborele de moșteniri
- O interfață este ca o clasă abstractă 100%, și implicit este declarată abstractă
- Interfețele pot avea doar metode abstracte. Începând cu Java 8 sunt permise și metode concrete cu comportament implicit (*default*) și metode statice
- Metodele din interfețe sunt implicit publice și abstracte. Menționarea acestor modificatori este optională
- Interfețele permit declararea constantelor, care implicit au modificatorii: public, static și final. Declararea explicită a acestor modificatori este optională



# Sumar interfețe: Implementarea interfețelor

- O clasă concretă care implementează o interfață are următoarele proprietăți:
  - Furnizează o implementare concretă a metodelor interfeței
  - Trebuie să respecte toate regulile de suprascriere a metodelor pe care le implementează
  - Permite tipul returnat să fie covariant
  - Nu poate declara noi excepții care nu au fost specificate în antetul metodei definite în interfață
- O clasă care implementează o interfață poate fi ea însăși abstractă; ea nu trebuie să implementeze neapărat metodele din interfață, însă lucrul acesta trebuie facut în prima subclasă concretă
- O clasă poate moșteni o singură clasă, dar poate implementa oricâte interfețe
- O interfață poate extinde una sau mai multe interfețe
- Interfețele nu pot extinde o clasă sau implementa vreo interfață



# Interfete în Java 7

- Java 7 sau versiuni mai vechi
  - Interfața poate avea 2 componente
    - Constante
    - Metode abstracte
  - Clasele concrete care implementează interfața **trebuie** să implementeze toate metodele definite de interfață
- Exemplu:

```
public interface DemoInterface {  
    public void div(int a, int b);  
}  
  
public class Demo implements DemoInterface{  
    public void div(int a, int b)  
    {  
        System.out.print("Metoda div: ");  
        System.out.println(a / b);  
    }  
  
    public static void main(String[] args)  
    {  
        DemoInterface x = new Demo();  
        Demo y = new Demo();  
        x.div(4,2);  
        y.div(30,7);  
    }  
}
```



# Interfete în Java 8

- Începând cu Java 8
  - Interfața poate avea 4 componente
    - Constante
    - Metode abstracte
    - Metode *default*
    - Metode statice
  - Clasele concrete care implementează interfața pot să suprascrie comportamentul implicit al metodelor *default*
    - Nu este obligatoriu!



# Interfete în Java 8

## ■ Exemplu:

```
public interface DemoInterface {  
  
    public void div(int a, int b);  
  
    public default void suma(int a, int b) {  
        System.out.print("Metoda default: ");  
        System.out.println(a + b);  
    }  
  
    public static void mul(int a, int b) {  
        System.out.print("Metoda statica: ");  
        System.out.println(a * b);  
    }  
}
```

```
public class Demo implements DemoInterface {  
  
    public void div(int a, int b) {  
        System.out.print("Metoda div: ");  
        System.out.println(a / b);  
    }  
  
    public static void main(String[] args) {  
        DemoInterface x = new Demo();  
        Demo y = new Demo();  
        y.div(4,2);  
        y.div(30,7);  
  
        DemoInterface z = new Demo();  
        z.div(8, 2);  
        z.suma(3, 2);  
        DemoInterface.mul(4, 9);  
    }  
}
```



TECHNICAL UNIVERSITY



# Programare orientată pe obiecte

1. Dezvoltarea aplicațiilor OO
2. Diagrame UML de clase și obiecte

Computer Science



# Proiectarea orientată pe obiecte

1. Descoperim clasele
2. Determinăm responsabilitățile fiecărei clase
3. Descriem relațiile dintre clase



# Unified Modeling Language (UML)

- UML este notația internațională standard pentru analiza și proiectarea orientată pe obiecte
- UML 2.0 definește treisprezece tipuri de diagrame, împărțite în două categorii:
  - **Diagrame de structură:** șase tipuri de diagrame reprezintă structura statică a aplicației:
    - **Diagrama de clase, diagrama de obiecte,** diagrama de componente, diagrama de structură compozită, diagrama de pachete și diagrama de desfășurare sistematică
  - **Diagrame de comportament:** șapte diagrame ce reprezintă tipuri generale de comportament:
    - Diagrama de activități, diagrama de interacțiune, diagrama cazurilor de utilizare (use case), diagrama de secvențe, diagrama de stare, diagrama de comunicare, diagrama de timp



# Descoperirea claselor

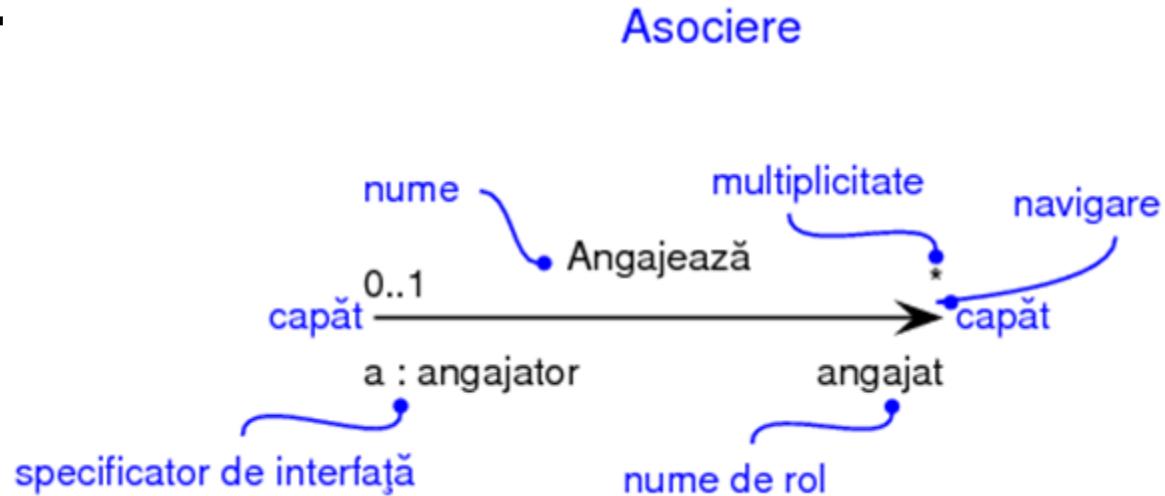
- O clasă reprezintă un concept util
  - Entități concrete: conturi bancare, elipse, produse, ...
  - Concepte abstracte: fluxuri (streams), ferestre grafice, ...
- Găsim *clasele* căutând *substantive* în descrierea sarcinii
- Definim comportamentul fiecărei clase
- Găsim *metodele* căutând *verbe* în descrierea sarcinii



# Relații între entitățile reprezentate

Tipuri de relații:

- Asociere
  - Agregare
  - Compoziție
- Dependență
- Generalizare
- Realizare (sau Implementare)

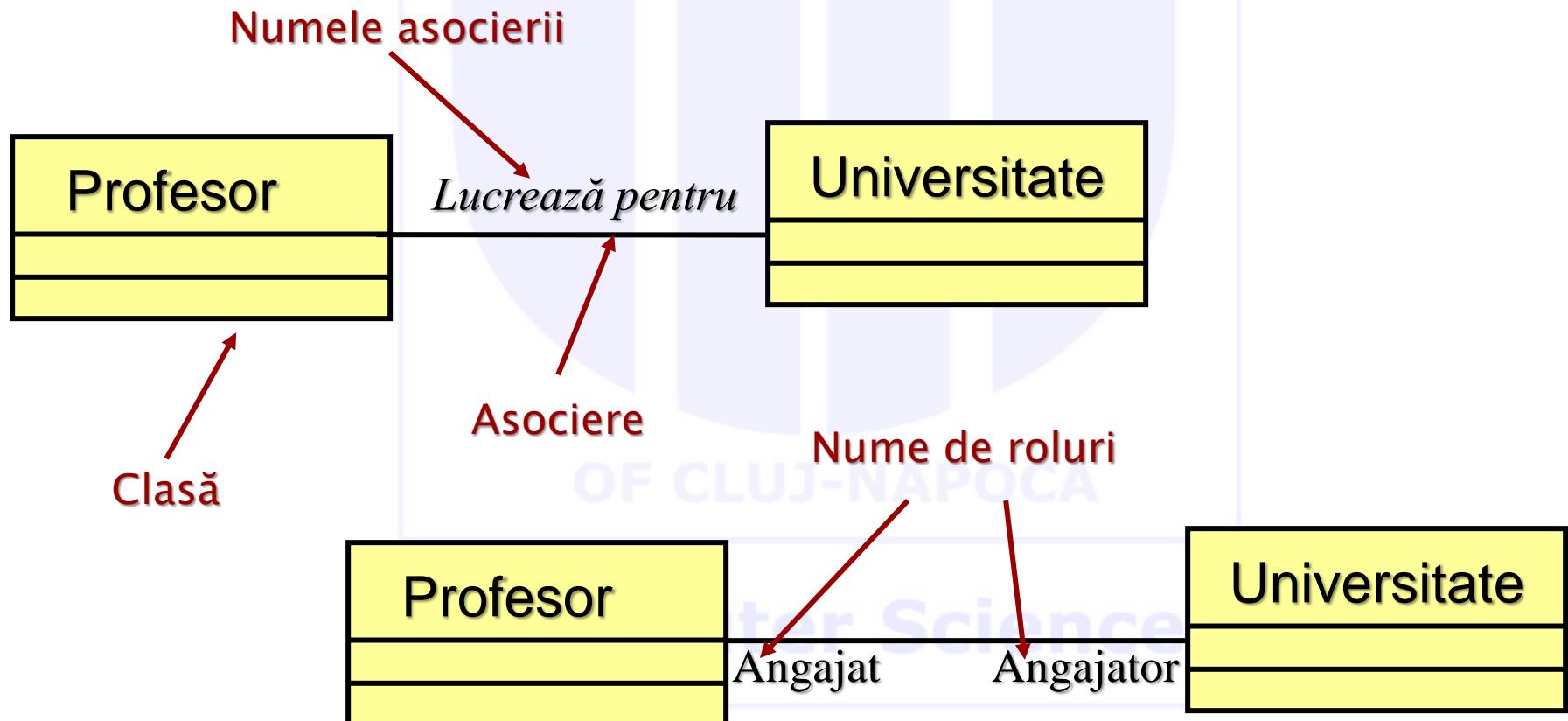


Computer Science



# Relații: Asociere

- Modeleză o conexiune semantică între clase





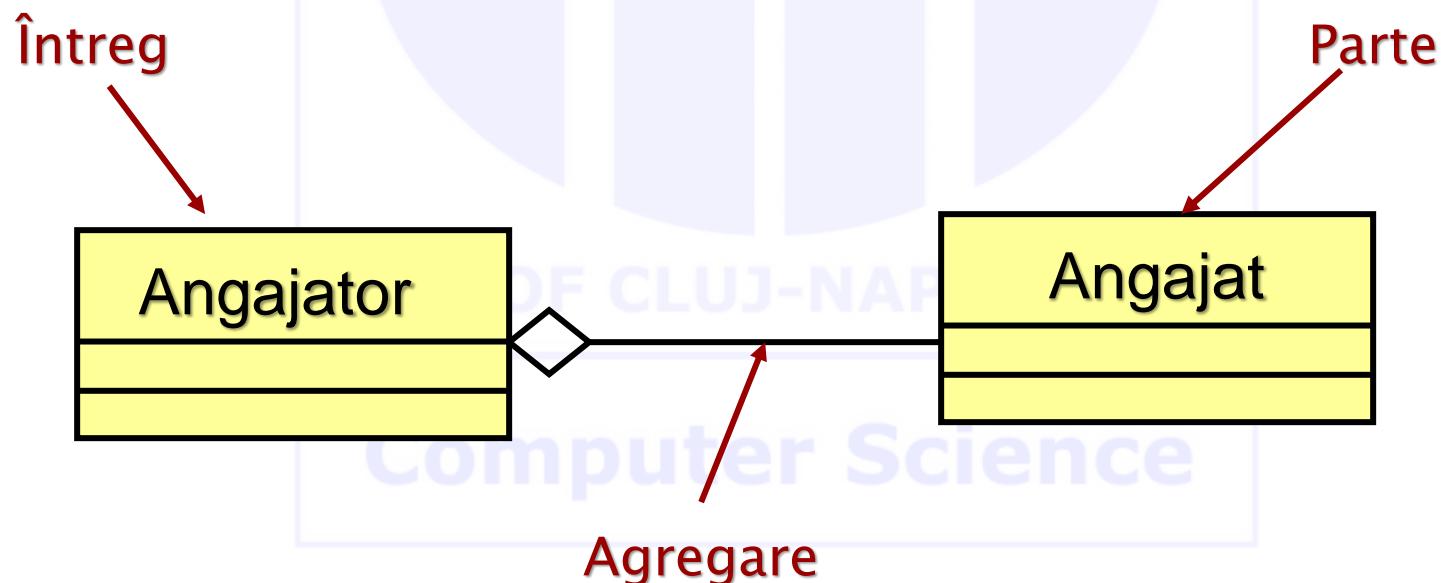
# Folosirea asocierilor

- Trei scopuri generale. Pentru a reprezenta:
  - O situație în care un obiect de o clasă *folosește* serviciile unui alt obiect, sau ele își folosesc reciproc serviciile – adică un obiect îi trimit mesaje celuilalt sau își trimit mesaje între ele
    - În primul caz, navigabilitatea poate fi unidirectională; în cel de al doilea, ea trebuie să fie bidirectională
  - *Agregarea* sau *compoziția* – unde *obiecte de o clasă sunt întregi compuși din obiecte de cealaltă clasă ca părți*
    - În acest caz, o relație de tip “folosește” este implicit prezentă – întregul folosește părțile pentru a-și îndeplini funcția, iar părțile pot și ele avea nevoie să folosească întregul
  - O situație în care obiectele sunt *înrudite*, chiar dacă nu schimbă mesaje
    - Aceasta se întâmplă de obicei când cel puțin unul dintre obiecte este folosit în esență la stocare de informație



# Relații: Agregare

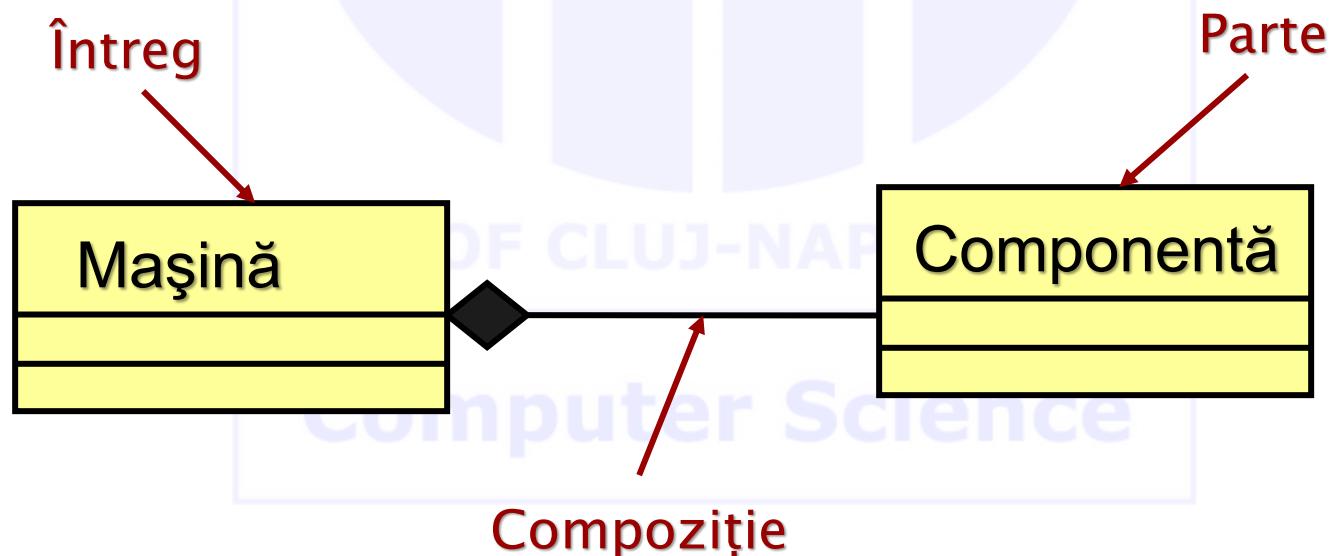
- Relație de tip: "are o/un"
- O formă specială de asociere care modelează relația parte–întreg între un agregat (întregul) și părțile sale





# Relații: Compoziție

- Relație de tip: "este parte a"
- O formă de agregare cu posesiune *puternică* și durată de viață care coincid
  - Părțile nu pot supraviețui fără existența întregului/agregatului





# Asociere: Multiplicitate și navigare

- Multiplicitatea definește câte obiecte participă într-o relație
  - Numărul de instanțe ale unei clase în raport cu o instanță a celeilalte clase
  - Specificat pentru fiecare capăt al asocierii
- Asocierile sunt implicit bidirectionale, dar adesea este de dorit să se restrângă navigarea la o singură direcție
  - Dacă navigarea este restricționată, se adaugă o săgeată pentru a indica direcția de navigare



# Asociere: multiplicitate

- Nespecificată

---

- Exact una

---

1

---

- Zero sau mai multe (multe, nelimitat)

0..\*

---

\*

- Una sau mai multe

---

1..\*

---

- Zero sau una

---

0..1

---

- Gama specificată

---

2..4

---

- Game multiple, disjuncte

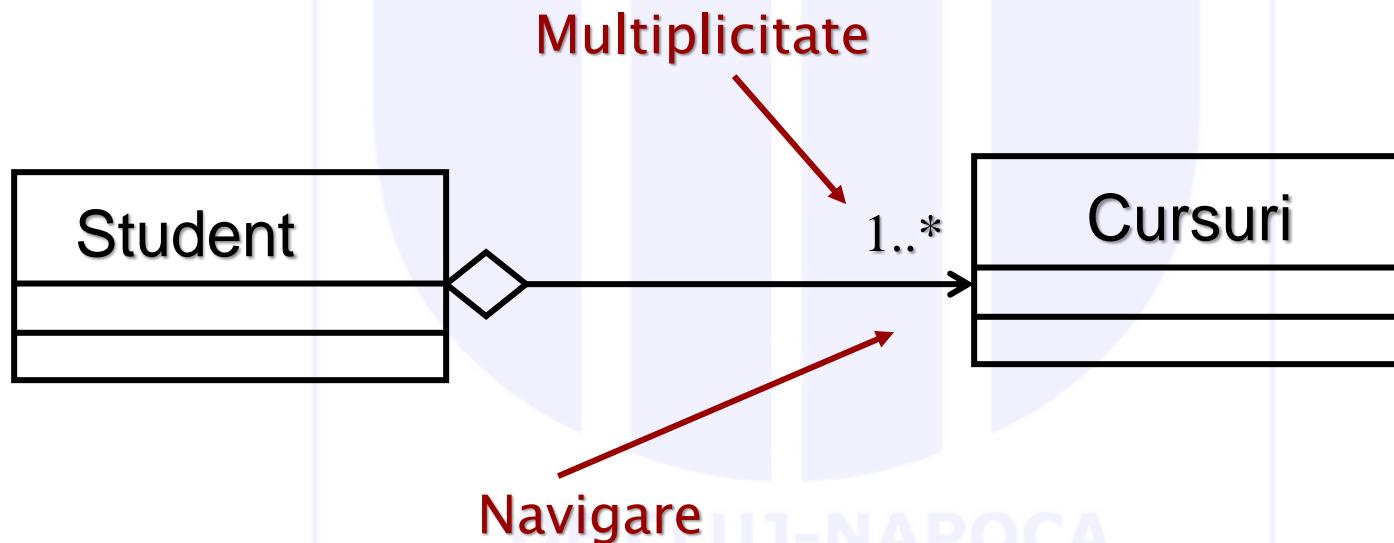
---

2, 4..6

Computer Science



# Exemplu: multiplicitate și navigare

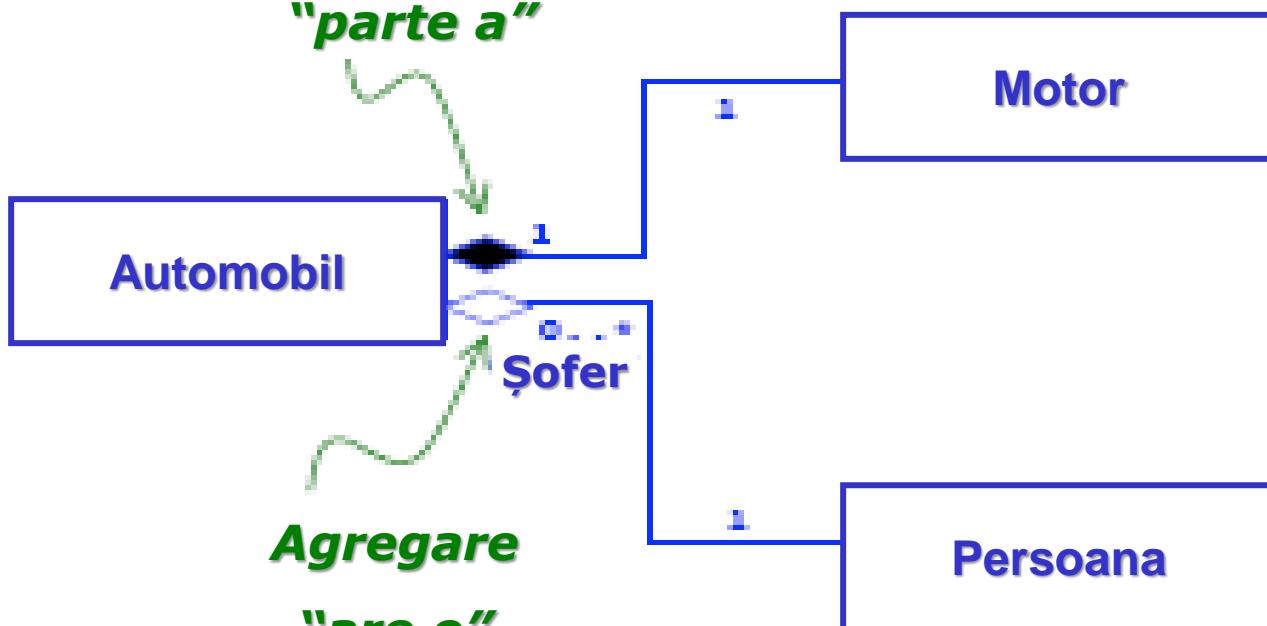




# Exemple de asociieri

**Compoziție**

**"parte a"**



**Agregare**

**"are o"**



# Observații

- **Teste pentru relații parte-întreg adevărate**
  - **Tranzitivitate:** dacă “A este parte a lui B” și “B este parte a lui C” atunci “A este parte a lui C”
    - Unghia este parte a degetului, degetul este parte a mâinii; atunci unghia este parte a mâinii
  - **O problemă a unei părți este o problemă a întregului**
    - O rană la unghie este o rană a mâinii
    - Pozițiile sunt parte a sistemului electric al automobilului. Un defect al pozițiilor este un defect al automobilului
- **Este-partea *e diferit* de**
  - **Este-Continut-În:** camăși, pantaloni,... --- dulap (observați că testul de defectare nu ține aici: pantalonii defecti nu înseamnă că dulapul e defect)
  - **Este-Legat-De:** dulap... --- persoană (care îl posedă)
  - **Este-Ramură-A:** artera iliacă,... --- aorta
  - **Se-Află-În:** casă... --- stradă



# Când să folosim agregarea

- **Ca regulă generală, se poate marca o asociere ca agregare, dacă sunt adevărate următoarele:**
  - Se poate spune că
    - Părțile 'sunt parte' a agregatului  
sau
    - Agregatul 'este compus din' părți
  - Când ceva deține sau controlează agregatul, atunci acel ceva deține sau controlează părțile



# Asociere, agregare și compozitie

## Asociere

**Obiectele știu unul despre altul astfel încât pot lucra împreună**

## Agregare

- Protejează integritatea configurației
- Funcționează ca un singur tot
- Controlul se face printr-un obiect – propagarea este în jos

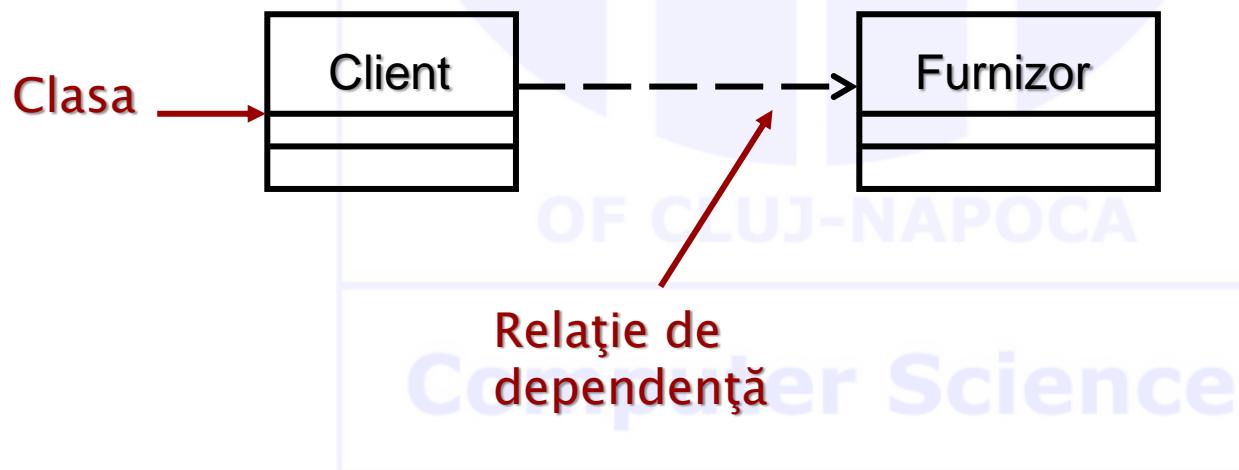
## Compoziție

**Fiecare parte poate fi membru al unui singur obiect aggregat**



# Relații: dependentă

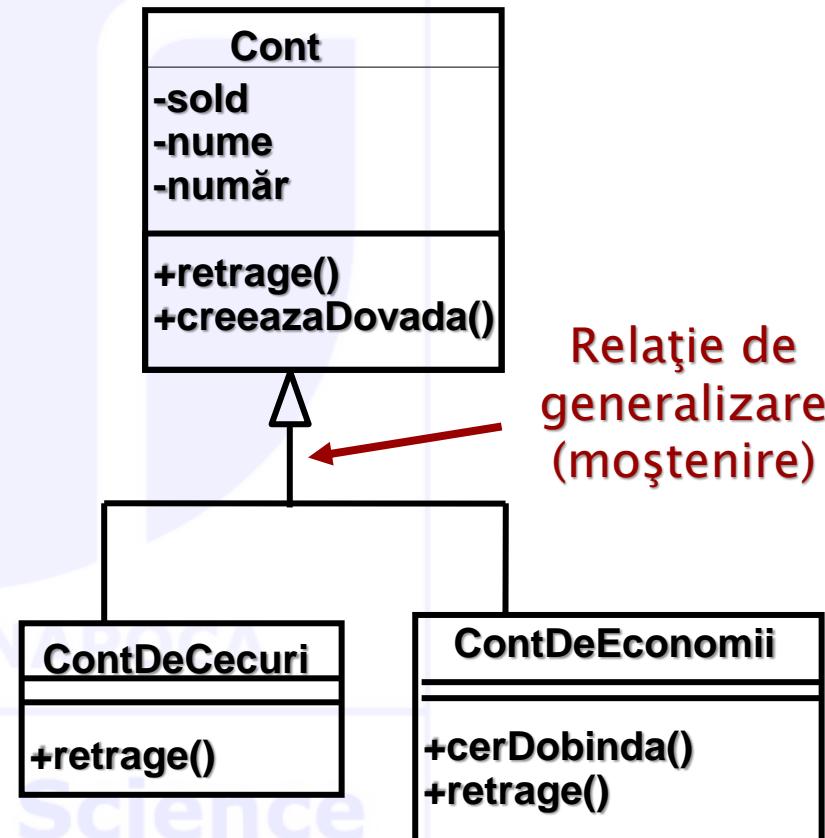
- Relație de tip "*folosește*"
- O relație între două elemente ale modelului în care o schimbare în unul dintre elemente **poate** cauza o schimbare în celălalt





# Relații: generalizare (sau moștenire)

- Relație de tipul *este-o/un*
- Relație între o clasă mai generală (superclasă) și una mai specializată (subclasă)
- Exemple:
  - Orice cont de economii este un cont bancar
  - Orice cerc este o elipsă (cu lățime și înălțime egale)





# Contra-exemplu de moștenire

- Uneori se abuzează de această relație
  - Ar trebui să fie clasa **Anvelopa** o subclasă a lui **Cerc**?
  - Relația *are-un* (**agregare**) ar fi mai potrivită aici
- Obiectele unei clase conțin referințe la obiectele altor clase
- Folosesc variabile instanță
  - O anvelopă are un cerc pe post de contur:

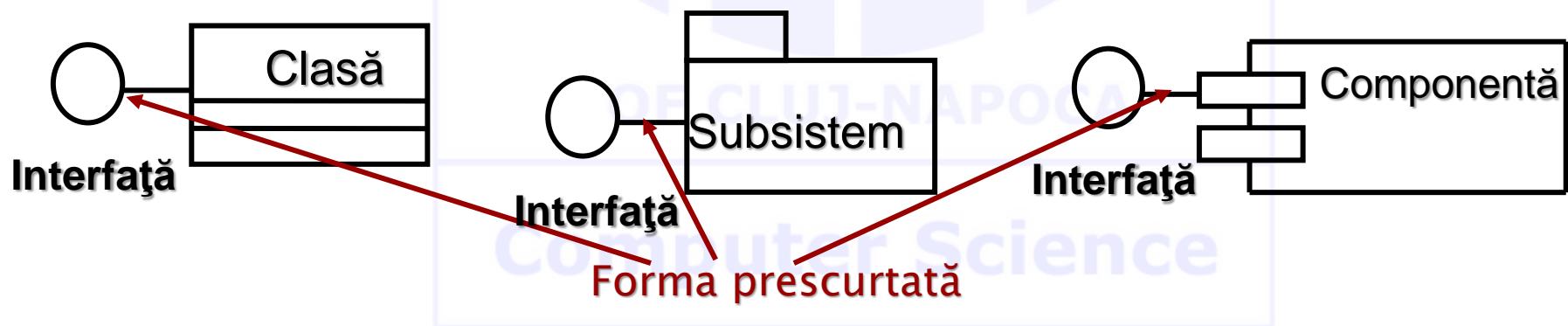
```
class Anvelopa {  
    . . .  
    private String catalogare;  
    private Cerc contur;  
}
```
- Fiecare automobil are mai multe anvelope

```
class Automobil extends Vehicul{  
    . . .  
    private Anvelopa[] anvelope;  
}
```



# Relații: realizare/ implementare

- O interfață servește pe post de contract pe care celălalt este de acord să-l îndeplinească
- Regăsit între:
  - Interfețele și clasele care le realizează (implementează)





# Note (adnotări)

- Se poate adăuga o notă (adnotare) la orice element UML
- Notele se pot adăuga pentru a furniza informații suplimentare în diagramă
- Este un dreptunghi cu colțul dreapta sus îndoit
- Nota poate fi ancorată la un element cu o linie îintreruptă

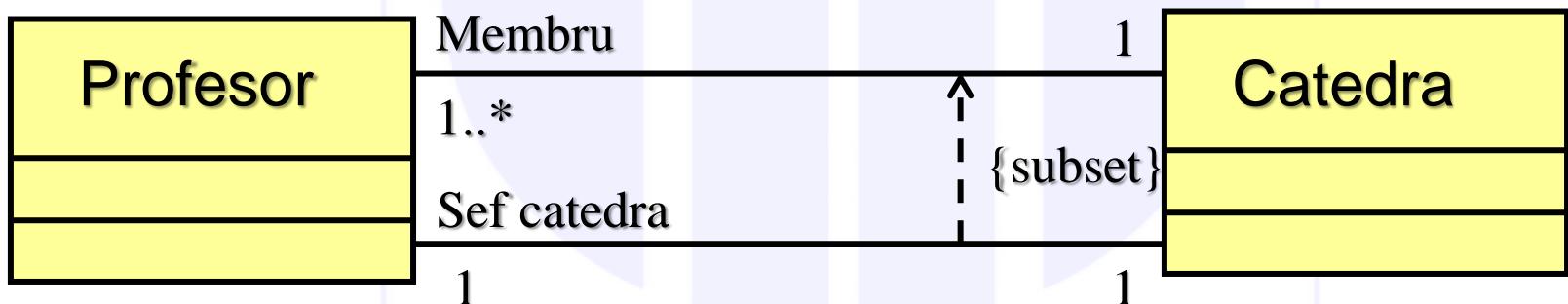


Poate exista cel mult un obiect  
IntretineFormularDePlanificare  
pentru o sesiune utilizator.



# Constrângeri

- Suportă adăugarea de reguli noi sau modificarea regulilor existente



- Această notație este folosită pentru a surprinde două relații între obiecte de tip Profesor și obiecte de tip Catedra, unde una dintre relații este un subset al celeilalte
- Arată cum se pot croi diagramele UML pentru a modela corect relații exacte



# Cartela CRC (*Class-Responsibility-Collaboration*)

- Cartela CRC: descrie o clasă, responsabilitățile și colaboratorii săi
- Se folosește o cartelă pentru fiecare clasă
- Alegem clasa care trebuie să fie răspunzătoare de fiecare metodă (verb)
- Scriem responsabilitatea pe cartela clasei
- Indicăm ce alte clase sunt necesare pentru a îndeplini responsabilitatea (colaboratorii)



# Cartela CRC (*Class-Responsibility-Collaboration*)

## ■ Cartela CRC

Numele clasei	
Responsabilități	Colaboratori

## ■ Un exemplu de cartelă CRC vs. diagramă de clase

Class CardReader	
Responsibilities	Collaborators
Tell ATM when card is inserted	ATM
Read information from card	Card
Eject card	
Retain card	

CardReader
- atm: ATM
+ <u>CardReader(atm: ATM)</u>
+ <u>readCard(): Card</u>
+ <u>ejectCard()</u>
+ <u>retainCard()</u>



# Simboluri UML pentru relații

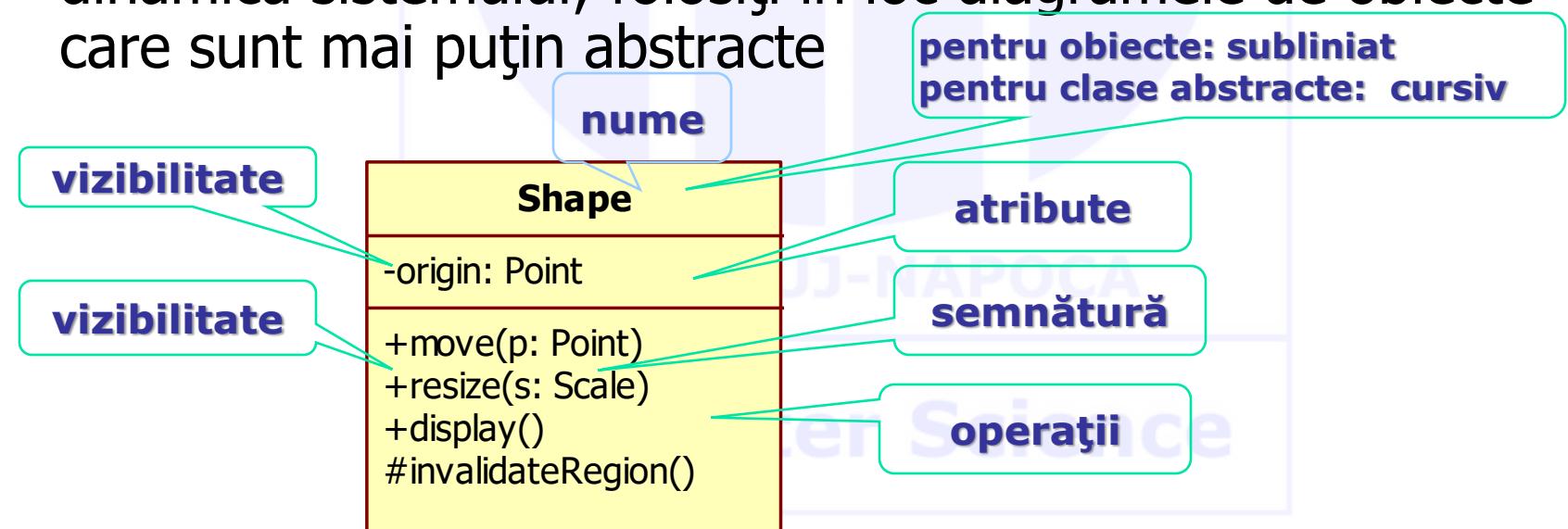
Relație	Simbol	Stil de linie	Forma vârfului
Moștenire		Solid	Triunghi
Implementare de interfață		Întrerupt	Triunghi
Agregare		Solid	Romb
Dependență		Întrerupt	Deschisă

Computer Science



# Diagramă de clase

- Reprezintă un set de clase, interfețe, colaborări și alte relații
- Reflectă *proiectul static* al unui sistem
- Poate genera confuzii dacă este folosit pentru a explica dinamica sistemului; folosiți în loc diagramele de obiecte care sunt mai puțin abstrakte





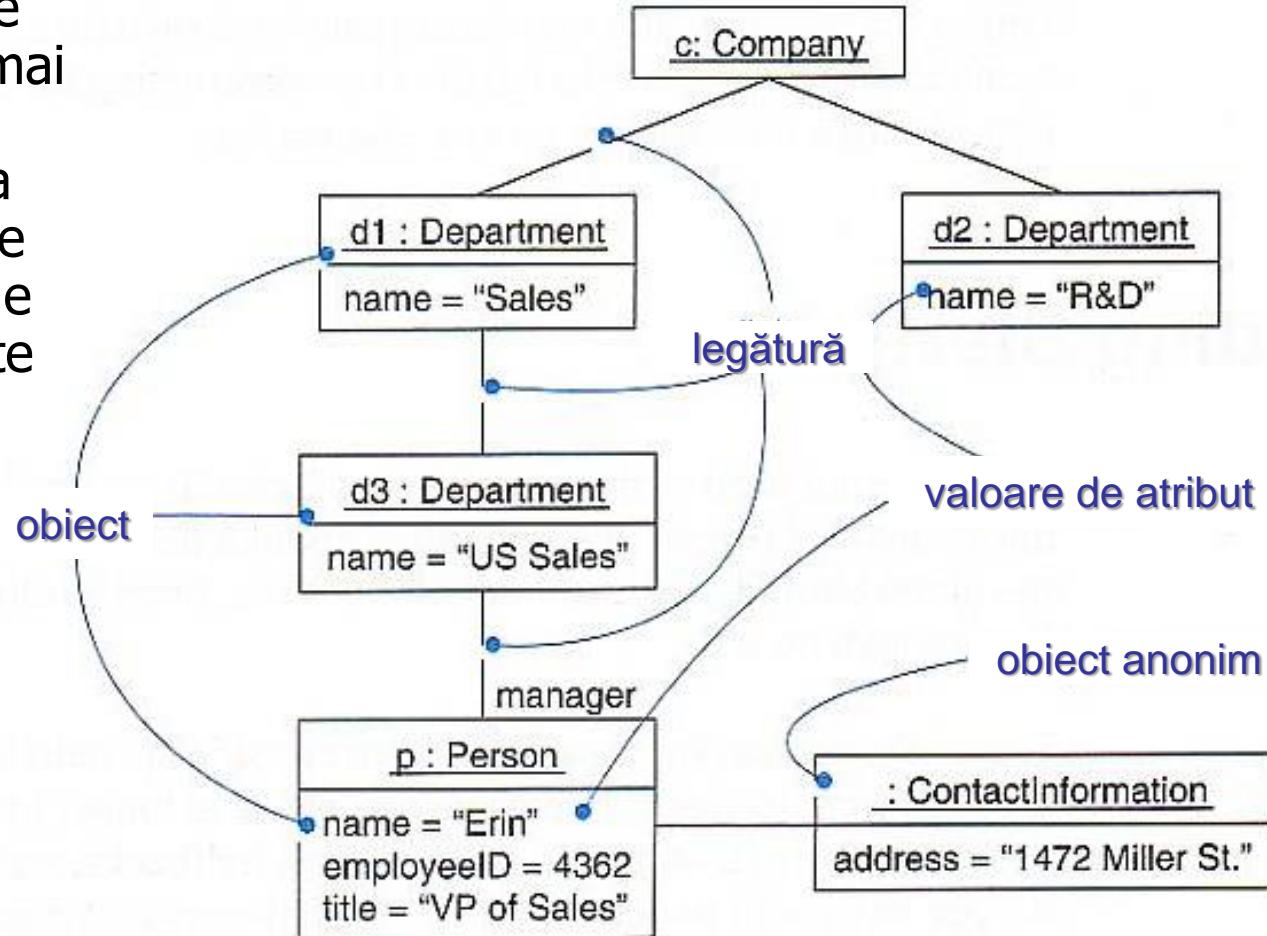
# Diagrame de obiecte

- Reprezintă un set de obiecte (instanțe de clase) și relațiile dintre acestea
- Este o *vedere dinamică* a sistemului la un moment dat
- Reprezintă cazuri reale sau cazuri prototip
- Foarte utile înainte de dezvoltarea diagramelor de clase



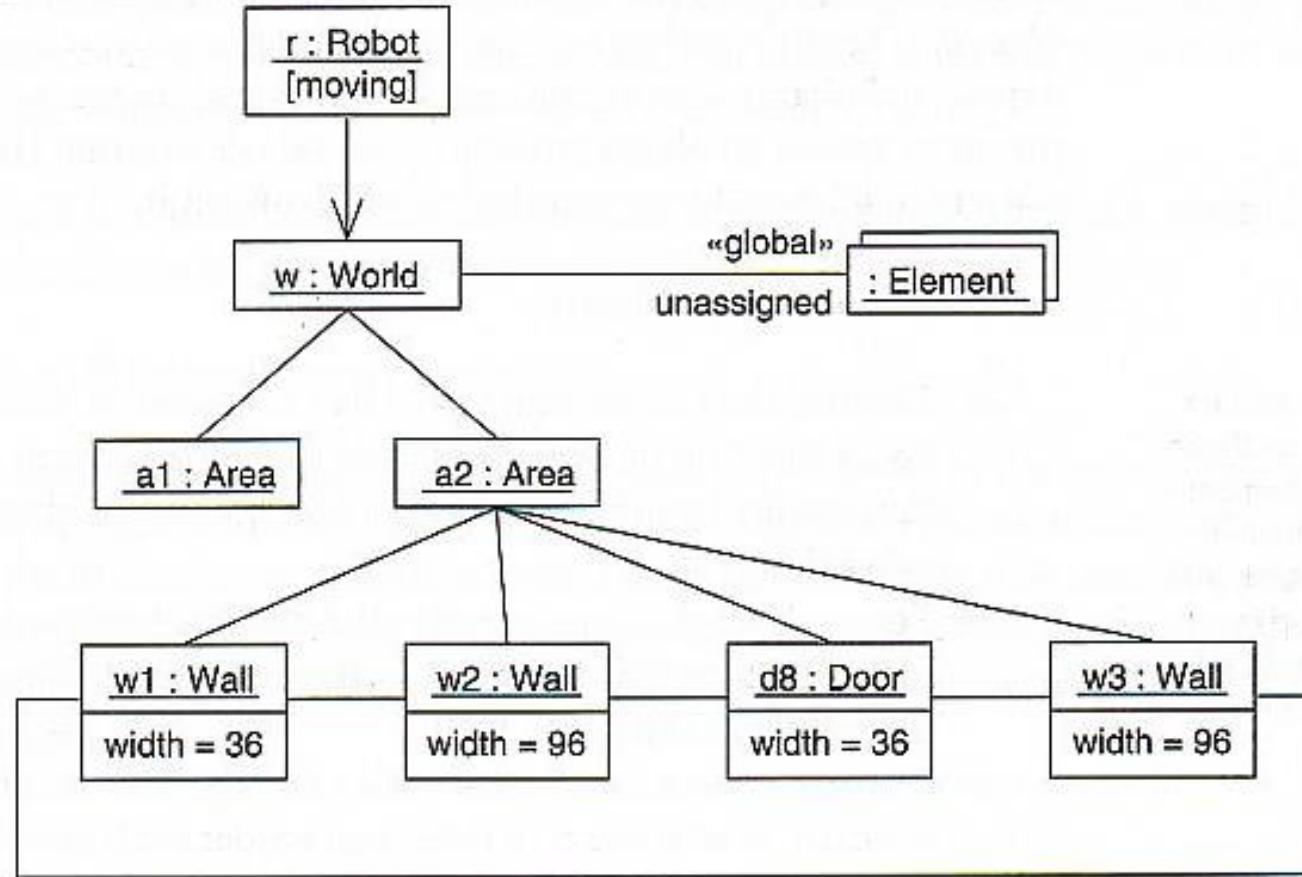
# Diagramă de obiecte: exemplu

- Diagrama de obiecte de mai jos este o instanțiere a diagramei de clase (clasele fiind înlocuite de obiecte concrete)





# Diagramă de obiecte: un alt exemplu





# Procesul de dezvoltare a unei aplicații OO în cinci pași

- Colectăm cerințele
- Folosim cartele CRC pentru a determina clasele, responsabilitățile și colaboratorii
- Folosim diagrame UML pentru a înregistra relațiile dintre clase
- Folosim **javadoc** pentru a documenta comportamentul metodelor
- Implementăm programul



# Reguli pentru determinarea claselor

- Între 3 și 5 responsabilități pe clasă
- Nu există clase singuratice
- Feriți-vă de multe clase mici
- Feriți-vă de puține clase mari
- Feriți-vă de “functoizi” – un functoid este de fapt o funcție procedurală normală deghizată în clasă
- Feriți-vă de clase omnipotente
- Evitați arborii de moștenire adânci

Computer Science



# Exemplu: factură simplificată

## FACTURA

Maria s.r.l.  
str. Mare nr. 1  
Un oraș, 554400

Descriere	Preț unitar	Cantitate	Total
Spray XXL	8.99	3	26.97
Şerveţele Super	2.99	4	11.96
Caiet studenţesc	3.99	2	7.98
Suma de plătit: 46.91			



## Exemplu: factură simplificată

- Clase care vin în minte: **Factura**, **Rînd**, și **Client**
- Este o idee bună să păstrăm o listă de clase candidate
- Folosim brainstorming-ul, pur și simplu punem toate ideile de clasă în listă
- Le putem tăia pe cel inutile ulterior



# Determinarea claselor

- Trebuie să încălziți minte următoarele puncte:
  - Clasele reprezintă mulțimi de obiecte cu același comportament
    - Entitățile cu apariții multiple în descrierea problemei sunt candidați buni pentru obiecte
    - Aflați ce au în comun
    - Proiectați clasele pentru a surprinde ce este comun
  - Reprezentați unele entități ca obiecte, iar altele ca tipuri primitive
    - Ar trebui să facem Adresa o clasă sau să folosim un String?
    - Nu toate clasele pot fi descoperite în faza de analiză
    - Unele clase pot exista deja



# Tipărirea unei facturi – cerințe

- Sarcina: tipărirea unei facturi
- Factura: descrie prețurile pentru un set de produse în anumite cantități
- Omitem lucrurile mai complicate – aici
  - Date, taxe și codurile de factură și de client
- Tipărim factura cu
  - Adresa clientului, toate rândurile, suma de plătit
- Rândul conține
  - Descriere, preț unitar, cantitatea comandată, prețul total
- Pentru simplitate nu creăm interfața cu utilizatorul
- Programul de test: adaugă rânduri în factură și apoi o tipărește



# Tipărirea unei facturi – cartele CRC

- Descoperim clasele
- Substantivele identifică clasele posibile

**Factura**  
**Adresa**  
**Rînd**  
**Produs**  
**Descriere**  
**Preț**  
**Cantitate**  
**Total**  
**Suma de plătit**



# Tipărirea unei facturi – cartele CRC

- Analizăm clasele

**Factura**

**Adresa**

**Rînd** // Înregistrează produsul și cantitatea

**Produs**

**Descriere** // Câmp al clasei Produs

**Pret** // Câmp al clasei Produs

**Cantitate** // Nu este un atribut al unui Produs

**Total** // Calculat, nu se memorează

**Suma de plătit** // Calculată, nu se memorează

- Clasele după un proces de eliminare

**Factura**

**Adresa**

**Rînd**

**Produs**



# Motive pentru respingerea unei clase candidate

Semnal	Motiv
<b>Clasă cu nume verb (infinitiv sau imperativ)</b>	Poate fi o subrutină, nu o clasă
<b>Clasă cu o singura metodă</b>	Poate fi o subrutină, nu o clasă
<b>Clasă descrisă că "efectuează" ceva</b>	Poate să nu fie o abstracțiune propriu-zisă
<b>Clasă fără metode</b>	Poate fi o informație opacă
<b>Clasă cu zero sau foarte puține atribute (dar care moștenește de la părinti)</b>	Poate fi un caz de exagerare în crearea de clase noi într-o taxonomie
<b>Clasă care acoperă câteva abstracțiuni</b>	Ar trebui divizată în mai multe clase, câte una pentru fiecare abstracțiune



# Cartelele CRC pentru tipărirea unei facturi

- Atât **Factura** cât și **Adresa** trebuie să se poată autoformată – **responsabilități**:
  - **Factura** *formatează factura și*
  - **Adresa** *formatează adresa*
- Adăugăm colaboratori pe cartela facturii: **Adresa** și **Rînd**
- Pentru cartela **Produs** – **responsabilități**: *obține descrierea, obține prețul unitar*
- Pentru cartela **Rînd** – **responsabilități**: *formatează articolul, obține prețul total*



# Cartelele CRC pentru tipărirea unei facturi

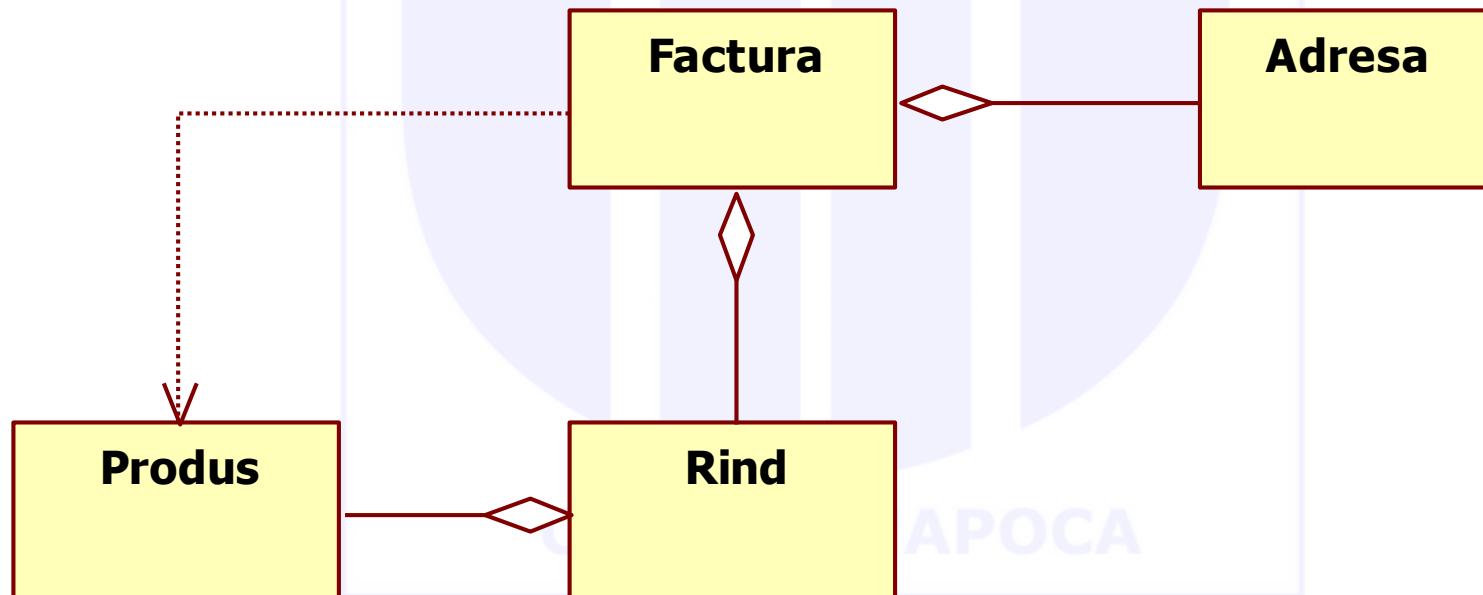
- **Factura** trebuie populată cu produse și cantități

Factura		
<i>formatează factura adăugă un produs și o cantitate</i>		<i>Adresa Rind Produs</i>

Computer Science



# Tipărirea unei facturi – diagrama UML



Computer Science



# Instrumente pentru realizarea diagramelor UML

- ArgoUML:
  - <https://argouml.en.softonic.com/>
  - rulează pe orice platformă Java
- WhiteStarUML
  - <http://sourceforge.net/projects/whitestaruml/>
  - proiect "open source"
- Poseidon for UML Community Edition
  - <http://www.gentleware.com/products.html>
- IBM Rational Modeler:
  - <http://www-03.ibm.com/software/products/fi/ratimode>
- Direct online:
  - <https://www.gliffy.com>
  - <https://www.draw.io/>



# Alte exemple de dezvoltare de aplicații OO

- Carte de adrese
  - <http://www.math-cs.gordon.edu/courses/cs211/AddressBookExample/>
- Automat bancar (Automatic Teller Machine)
  - <http://www.math-cs.gordon.edu/courses/cs211/ATMExample/>

Computer Science



# Programare orientată pe obiecte

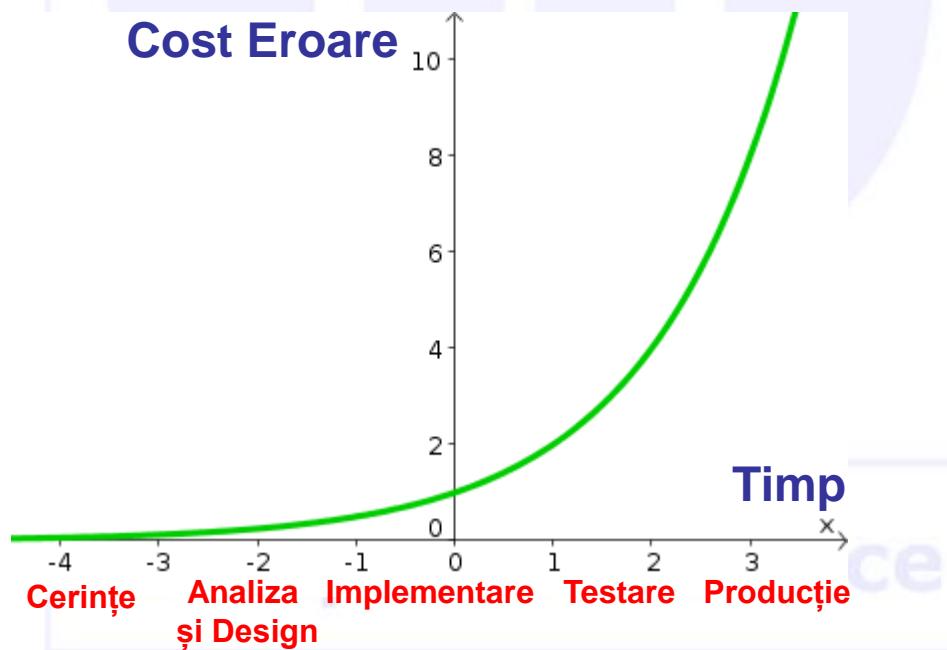
## 1. Erori și exceptii în Java

Computer Science



# De ce mecanisme de tratare a erorilor?

- Costul de remediere a unei erori nerezolvate poate crește exponențial odată cu timpul:





# Soluții în Java

- Soluții
  - Aserțiuni
  - Mecanisme de tratare a exceptiilor
  - Mecanisme de testare – Testare Unitară



# Aserțiuni

- Aserțiunile sunt niște mecanisme ce permit testarea logicii programului în **faza de dezvoltare**
- În **faza de producție** asertiunile se dezactivează, programul rulând normal fără a ține cont de existența aserțiunilor
- Avantajul principal al folosirii aserțiunilor
  - Timpul de implementare / testare scăzut – deoarece în unele cazuri nu merită efortul scrierii de cod pentru tratarea unor excepții care în faza de producție se știe că nu au cum să apară
- Raționament:
  - Presupunem (**assert**) totdeauna că o anumită condiție este adevărată, ex:  
`assert (x>y)`
  - În cazul în care condiția nu este îndeplinită, programul va arunca o **AssertionError**



# Aserțiuni

## ■ Sintaxa:

- `assert expr1;`

sau

- `assert expr1 : expr2`

`expr1` – expresie booleană,  
valoarea “false” indică o eroare

`expr1` – expresie booleană  
`expr2` – valoare, mesajul erorii



# Activarea/dezactivarea aserțiunilor

- Activare:

```
java [ -enableassertions | -ea ] [:<package name>"..." | :<class name> ]
```

- Dezactivare:

```
java [ -disableassertions | -da ] [:<package name>"..." | :<class name> ]
```

- Observație: În alte medii de dezvoltare, opțiunea de utilizare a aserțiunilor trebuie activată/dezactivată explicit
  - În Eclipse opțiunile de mai sus sunt introduse prin accesarea Run Configurations -> tab-ul Arguments -> căsuța "VM Arguments", unde se inserează -ea sau -da



# Aserțiuni: exemplu simplu

```
public static void main(String[] args) {  
  
    String result = null;  
  
    /*  
     * ... Cod calcul rezultat  
     */  
  
    //Se presupune ca nu este legal sa avem valori null  
    assert result != null : "Nu sunt acceptate valori null";  
  
    System.out.println("end");  
  
}
```

Computer Science



# Aserțiuni: când se folosesc

- Se folosesc aserțiuni doar pentru a verifica validitatea unor condiții care în mod normal nu ar avea cum să fie negative sub nici o formă!
- Nu folosiți aserțiuni de exemplu pentru a valida datele de intrare ale programului
  - Soluția oportună pentru o astfel de validare este folosirea mecanismelor de tratare a excepțiilor
  - În cazul în care datele de intrare nu sunt cele dorite, greșeala se poate trata cerând utilizatorului să reintroducă datele respective

Computer Science



# Unde se pun aserțiunile

## ■ Locuri posibile

- **Precondiția** metodei – se verifică ce trebuie să fie adevărat atunci când se execută o metodă
- **Postcondiția** metodei – ce trebuie să fie adevărat după ce s-a executat metoda
- **Invarianți interni** – presupuneri că anumite porțiuni de cod sunt adevărate tot timpul
- **Invariantul** clasei – ce trebuie să fie adevărat tot timpul legat de variabilele de instanță

Computer Science



# Așteptări: alte exemple

```
public int calculeazaLungimeString(String inString)
{
    //PRECONDITIE
    assert inString != null : "Nu sunt acceptate valori null";

    int lungime = -1;

    lungime = inString.length();
    /*
     * ... cod ...
     */

    //POSTCONDITIE
    assert lungime >= 0 : lungime + " < 0";

    return lungime;
}
```



# Aserțiuni: alte exemple

## ■ Invarianti interni

```
if (i % 3 == 0) {  
    ...  
} else if (i % 3 == 1) {  
    ...  
} else {  
    assert i % 3 == 2 : i;  
    ...  
}
```

## ■ Invarianti ai instructiunilor de control

```
void foo() {  
    for (...) {  
        if (...)  
            return;  
    }  
    assert false; /* Execuția nu ar trebui să ajungă în acest punct  
                   NICIODATA! */  
}
```



# Aserțiuni: alte exemple

- Invariantul clasei – proprietățile clasei nu se schimbă niciodată atât înainte cât și după execuția oricărei metode
  - Exemplu: clasa ce implementează un arbore binar echilibrat; un invariant al clasei ar fi că **arborele este echilibrat tot timpul**

- Se poate introduce o metodă ce testează dacă un arbore este echilibrat sau nu

```
private boolean balanced() {  
    ...  
}
```

- Fiecare metodă sau constructor public al clasei va trebui să conțină această constrângere imediat înainte de return

```
assert balanced();  
return ...;
```



# Exceptii. Probleme în cursul execuției programelor

- Un program întâlnește adesea probleme (**exceptii**) în cursul execuției sale:
  - poate avea probleme la citirea datelor,
  - pot exista caractere nepermise în date sau
  - indexul unui tablou poate depăși limitele acestuia
- Exceptiile Java permit programatorului să trateze astfel de probleme
  - Putem scrie programe care își revin la întâlnirea exceptiilor și își continuă execuția
  - **Programele nu trebuie să eșueze atunci când utilizatorul face o greșală!**
- În special intrarea și ieșirea sunt susceptibile la exceptii
- Tratarea exceptiilor este esențială pentru programarea I/E



# Exemplu de apariție a unei excepții

## ■ Programul:

```
import java.util.Scanner;  
  
public class InputMismatchExceptionDemo {  
    public static void main(String[] args) {  
        Scanner keyboard = new Scanner(System.in);  
        System.out.print("Enter one integer:");  
        int inputNumber = keyboard.nextInt();  
        System.out.println("The square of " + inputNumber + " is " + inputNumber *  
                           inputNumber);  
    }  
}
```

## ■ Cu intrarea: Enter one integer:h1

## ■ Are rezultatul:

```
java.util.InputMismatchException  
at java.util.Scanner.throwFor(Scanner.java:819)  
at java.util.Scanner.next(Scanner.java:1431)  
at java.util.Scanner.nextInt(Scanner.java:2040)  
at java.util.Scanner.nextInt(Scanner.java:2000)  
at InputMismatchExceptionDemo.main(InputMismatchExceptionDemo.java:11)
```



# Discuție asupra exemplului

- Programul nu este greșit
  - Problema este că `nextInt` nu poate converti sirul de caractere "h1" la un `int`
  - În momentul în care `nextInt` a întâlnit problema, metoda a **aruncat** o excepție de tipul `InputMismatchException`
  - Sistemul de execuție Java a interceptat (a "prins") excepția, a oprit programul și a tipărit mesajele de eroare



# Exceptii și erori

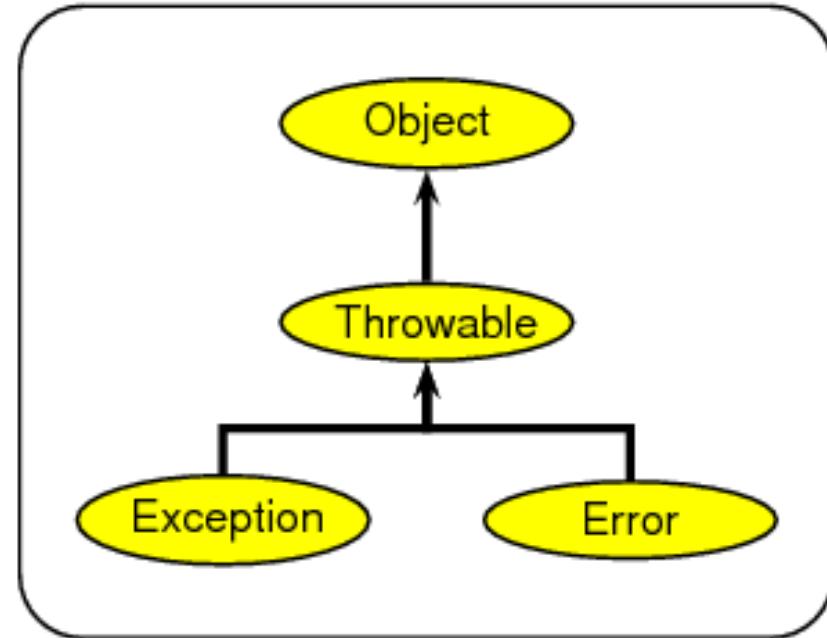
- O **excepție**: o problemă care apare în cursul execuției unui program
  - La apariția unei exceptii, JVM creează un obiect de clasa **Exception** care conține informații despre problema apărută
  - Însuși programul Java poate **intercepta (catch)** o excepție. Apoi poate folosi obiectul de tipul excepție pentru a-și reveni după problemă
- și o **eroare** este o problemă care apare la rularea unui program
  - O eroare este reprezentată de un obiect de clasa **Error**
  - Dar o eroare este prea severă pentru a fi tratată de un program. Programul trebuie să-și **înceteze execuția**

Computer Science



# Ierarhia Throwable ("aruncabil")

- Atât clasa **Exception** cât și clasa **Error** extind clasa **Throwable**
  - O metodă Java poate "arunca" un obiect de clasa **Throwable**
  - D.e. `Integer.parseInt("zzz")` aruncă o excepție atunci când încearcă să convertească "zzz" într-un întreg
- **Excepții != Erori:** se pot scrie programele astfel încât să-și revină după excepții, dar nu se pot scrie astfel încât să-și revină după erori





# Tratarea exceptiilor. Mecanismul **try-throw-catch**

- Calea fundamentală pentru tratarea exceptiilor în Java constă din trio-ul **try-throw-catch**
- Blocul **try** conține codul pentru algoritmul implementat
  - Acest cod spune ce se face atunci când totul merge bine
  - Se numește bloc **try** deoarece el "încearcă" să execute cazul în care totul merge aşa cum a fost planificat
  - De asemenea acest bloc poate conține cod care aruncă o excepție dacă se întâmplă ceva neobișnuit

```
try {  
    CodCarePoateAruncaOExceptie  
}
```



# Tratarea exceptiilor. Mecanismul **try-throw-catch**

- Aruncarea explicită a unei exceptii

```
throw new NumeleClaseiExceptie(PosibilArgumente);
```

- La aruncarea unei exceptii, execuția blocului **try** în care a fost aruncată exceptia se oprește
  - Normal, controlul este transferat unei alte porțiuni de cod, blocul **catch** (blocul de interceptare)
- Valoarea aruncată este argumentul operatorului **throw**; ea este întotdeauna un obiect aparținând unei clase exceptie
  - Execuția unei instrucțiuni **throw** se numește *aruncare a unei exceptii*



# Tratarea exceptiilor. Mecanismul **try-throw-catch**

- O instrucțiune **throw** seamănă cu un apel de metodă  
**throw new NumeClasaExceptie(UnString) ;**
  - În exemplul de mai sus, obiectul de clasa **NumeClasaExceptie** este creat folosind ca argument un sir de caractere
  - Acest obiect, care este argument pentru operatorul **throw**, este obiectul excepție aruncat
- În loc să apeleze o metodă, instrucțiunea **throw** apelează un bloc **catch**



# Tratarea exceptiilor. Mecanismul **try-throw-catch**

- La aruncarea unei exceptii se incepe executarea blocului **catch**
  - Blocul **catch** are *un parametru*
  - Obiectul exceptie aruncat este transmis ca parametru al blocului **catch**
- Un bloc **catch** este o portiune de cod separată care se execută atunci când un program întâlnește și execută o instrucțiune **throw** în blocul **try** precedent
  - Execuția blocului **catch** se numește *interceptarea/"prinderea"* *exceptiei*, sau *tratarea exceptiei*

```
catch(Exception e) {  
    CodDeTratareAExceptiei  
}
```



# Tratarea exceptiilor. Mecanismul **try-throw-catch**

```
catch (Exception e) { . . . }
```

- Identifierul **e** din blocul **catch** de deasupra se numește parametru al blocului **catch**
- Parametrul blocului **catch** îndeplinește două roluri:
  1. Specifică tipul de obiect excepție aruncat pe care blocul **catch** îl poate intercepta (d.e., mai sus este un obiect de clasa **Exception**)
  2. Oferă un nume (pentru obiectul care este interceptat) care să fie folosit în blocul **catch**
    - Observație: adesea se folosește identifierul **e** prin convenție, dar se poate folosi orice identifier care nu este cuvânt cheie



# Tratarea exceptiilor. Mecanismul **try-throw-catch**

- La executarea unui bloc **try** se pot întâmpla două lucruri:
  1. Nu este aruncată nici o excepție în blocul **try**
    - Codul din blocul **try** este executat până la sfârșitul blocului
    - Blocul **catch** este sărit
    - Execuția continuă de la codul amplasat după blocul **catch**
  2. Este aruncată o excepție în blocul **try** și interceptată în blocul **catch**
    - Restul codului din blocul **try** este sărit
    - Controlul se transferă la un bloc **catch** următor (în cazurile simple)
    - Obiectul aruncat este transmis ca parametru al blocului **catch**
    - Se execută codul din blocul **catch**
    - Se execută codul care urmează după blocul **catch** respectiv (dacă există)



# Blocuri **catch** multiple

- Un bloc **try** poate arunca potențial orice număr de valori excepție, iar acestea pot fi de tipuri diferite
  - În oricare execuție a unui bloc **try**, poate fi aruncată cel mult o excepție (de vreme ce instrucțiunea **throw** termină execuția blocului **try**)
  - La execuții diferite ale blocului **try** pot fi aruncate valori diferite
- Fiecare bloc **catch** poate intercepta valorile de tipul de clasă excepție, date în antetul blocului **catch**
- Se pot intercepta tipuri diferite de excepții punând mai multe blocuri **catch** după un bloc **try**
  - Se pot pune oricâte blocuri **catch**, dar în ordinea corectă

Computer Science

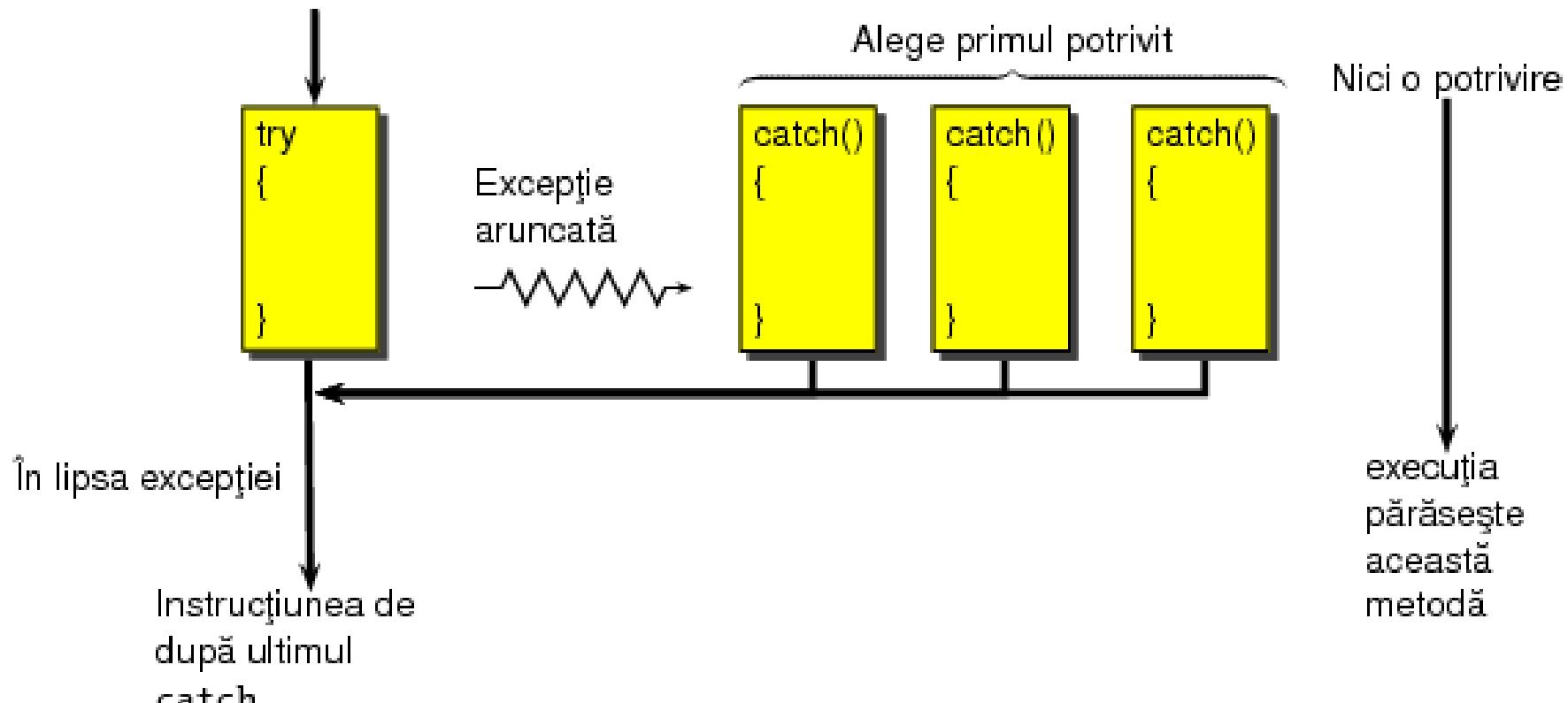


# Capcană: Interceptați mai întâi cea mai specifică excepție

- La interceptarea de excepții multiple, ordinea blocurilor **catch** este importantă
  - **La aruncarea unei excepții într-un bloc `try`, blocurile `catch` sunt examineate în ordinea aparitiei**
  - **Este executat primul bloc care se potrivește cu tipul de excepție aruncat**
- ```
catch (Exception e)
{ . . .
catch (NegativeNumberException e)
{ . . . }
```
- Deoarece **NegativeNumberException** este un tip de **Exception**, toate **NegativeNumberExceptions** vor fi interceptate de către primul bloc **catch** înainte de a ajunge vreodată la cel de-al doilea
  - **Blocul catch pentru `NegativeNumberException` nu va fi folosit!**
- Pentru ordine corectă, inversați cele două blocuri



# Tratarea excepțiilor. Mecanismul **try-throw-catch**





# Exemplu cu două exceptii

```
public class DublaGreseala {  
    public static void main(String[] args) {  
        int num = 5, denom = 0, result;  
        int[] arr = {7, 21, 31};  
        try  
        {  
            result = num / denom;  
            result = arr[num];  
        }  
        catch (ArithmetricException ex) {  
            System.out.println("Eroare aritmetica");  
        }  
        catch (IndexOutOfBoundsException ex) {  
            System.out.println("Eroare de indice");  
        }  
    }  
}
```

**Observație:**  
**Cea de a doua**  
**excepție nu va fi**  
**aruncată niciodată!**



# Tratarea exceptiilor. Mecanismul **try-throw-catch**

- La aruncarea unei exceptii de catre o instructiune blocul **try{}**, blocurile **catch{}** sunt examineate unul cate unul incepand cu primul
- Un singur bloc catch{} este ales
- Daca nici un bloc **catch{}** nu se potriveste cu exceptia, atunci nu este ales nici unul, iar execuția părăsește metoda respectivă (exact ca în lipsa blocului **catch{}**)
- Primul bloc **catch{}** care se potrivește cu tipul de excepție aruncată obține controlul
- Cele mai specifice tipuri de excepție trebuie să apară la început, urmate de tipurile mai generale de excepție
- Instructiunile din blocul **catch{}** ales sunt executate secvențial; după executarea ultimei instructiuni, controlul ajunge la prima instructiune care urmează după structura **try/catch**
- Controlul nu se întoarce în blocul **try**



# Exemplu de intrare "prietenoasă"

```
import java.lang.* ;
import java.io.* ;

public class SquareUser
{
    public static void main ( String[] a )
        throws IOException
    {
        BufferedReader stdin =
            new BufferedReader (
                new InputStreamReader(System.in));
        String  inData = null;
        int      num = 0;
        boolean inputOK = false;
        while ( !inputOK )
        {
            System.out.print("Introduceti un
                            intreg:");
            inData = stdin.readLine();

```

```
try
{
    num = Integer.parseInt( inData );
    inputOK = true;
}
catch (NumberFormatException ex )
{
    System.out.println("Ati introdus
                        date invalide.");
    System.out.println("Va rog sa
                        reincercati.\n");
}
//end while
System.out.println("Patratul lui " +
                    inData + " este " + num*num);
}
```



# Clauza `finally`

- Exceptia provoaca terminarea metodei curente
- Pericol: se poate sa scrie peste o portiune de cod esentiala
- Exemplu

```
reader = new FileReader(filename);
Scanner in = new Scanner(reader);
readData(in);
reader.close(); // s-ar putea sa nu ajunga aici niciodata
```

- Trebuie executat `reader.close()` chiar daca apare o exceptie
- Folosim clauza `finally` pentru codul care trebuie executat "indiferent de ce se intampla" (neconditioнат)



# Blocuri **catch** multiple și clauza **finally**

- Dacă există clauze **catch** asociate blocului **try**, atunci trebuie să punem clauza **finally** după toate clauzele **catch**

```
try {  
    // Bloc de cod cu puncte de ieșire multiple  
}  
catch (OneException e) {  
    System.out.println(" Am interceptat OneException!");  
}  
catch (OtherException e) {  
    System.out.println(" Am interceptat OtherException!");  
}  
catch (AnotherException e) {  
    System.out.println(" Am interceptat AnotherException!");  
}  
finally {  
    // Bloc de cod executat intotdeauna la ieșirea din bloc,  
    // indiferent de cum s-a ieșit din "try"  
    System.out.println("Finally este executat intotdeauna");  
}
```



# Clase excepție

- Există mai multe clase excepție pe lângă clasa **Exception**
  - Există mai multe clase excepție în bibliotecile standard Java
  - Pot fi definite noi clase excepție exact ca orice alte clase
- Toate clasele excepție predefinite au următoarele proprietăți
  - Posedă un constructor cu un singur argument de tipul **String**
  - Clasa are o metoda accesoare, **getMessage()**, care poate recupera sirul dat ca argument constructorului la crearea obiectului excepție
- Toate clasele excepție definite de programator ar trebui să aibă aceleași proprietăți



# Clase excepție din pachetele standard

- Există numeroase clase excepție predefinite care sunt incluse în pachetele standard Java:
  - `IOException`
  - `NoSuchMethodException`
  - `FileNotFoundException`
- Multe clase excepție trebuie importate pentru a le putea utiliza:  
`import java.io.IOException;`
- Clasa predefinită `Exception` este clasa rădăcină pentru toate excepțiile
  - Fiecare clasă excepție este descendenta din clasa `Exception`
  - Clasa `Exception` poate fi folosită: direct sau pentru a defini o clasă derivată
  - Clasa `Exception` se află în pachetul `java.lang` și nu trebuie clauză `import`



# Folosirea metodei `getMessage`

```
 . . . // codul metodei  
try  
{  
    . . .  
    throw new Exception(StringArgument);  
    . . .  
}  
catch(Exception e)  
{  
    String message = e.getMessage();  
    System.out.println(message);  
    System.exit(0);  
} . . .
```

- Fiecare excepție are o variabilă instanță de tipul **String** care conține un mesaj
  - Acest sir identifică de obicei motivul apariției excepției
- **StringArgument** este folosit ca valoare pentru variabila instanță de tip sir a excepției **e**
  - De aceea, apelul de metodă **e.getMessage()** returnează acest sir



# Definirea claselor excepție

- Fiecare clasă excepție care urmează să fie definită trebuie să fie o clasă derivată dintr-o clasă excepție deja definită
  - Derivată din oricare clasă excepție definită în bibliotecile standard Java sau definită de către programator
- Constructorii sunt membrii cei mai importanți în definirea unei clase excepție
  - Constructorii trebuie să se comporte corespunzător în raport cu variabilele și metodele moștenite din clasa de bază
  - Adesea, nu există alți membri cu excepția celor moșteniți din clasa de bază
- Clasa care urmează nu efectuează decât aceste sarcini fundamentale



# O clasă excepție definită de către programator

```
public class DivisionByZeroException extends Exception  
{  
    public DivisionByZeroException()  
    {  
        super("Division by zero.");  
    }  
    public DivisionByZeroException(String message)  
    {  
        super(message);  
    }  
}
```

/\* Se poate face mai mult într-un constructor de excepție, dar aceasta este o formă uzuală \*/

/\* super invocă constructorul clasei de bază Exception \*/

Computer Science



# Characteristicile obiectului Exception

- Cele mai importante două lucruri referitoare la un obiect excepție sunt tipul său (adică, clasa excepție) și mesajul pe care îl poartă
  - Mesajul este transmis împreună cu obiectul excepție ca variabilă instanță
  - Acest mesaj poate fi recuperat cu metoda accesoare `getMessage`, astfel că blocul `catch` poate folosi mesajul



# Indicații pentru clasele excepție definite de programator

- Clasele excepție pot fi definite de către programator, dar fiecare asemenea clasă trebuie să fie derivată dintr-o clasă excepție existentă deja
- Clasa **Exception** poate fi folosită pe post de clasă de bază, cu excepția cazului în care o altă clasă excepție este mai potrivită
- Trebuie definiți cel puțin doi constructori, iar uneori mai mulți
- Excepția trebuie să țină seama că metoda **getMessage()** este moștenită



# Să păstreze getMessage

- Pentru toate clasele excepție predefinite, **getMessage** returnează sirul de caractere transmis ca argument constructorului său
  - Sau să returneze un sir implicit dacă nu s-a transmis nici un argument constructorului
- Acest comportament trebuie păstrat în toate clasele excepție definite de către programator
  - Trebuie inclus un *constructor* care are *un parametru sir* de caractere și al cărui corp începe cu un apel la **super**
    - Apelul la **super** trebuie să folosească parametrul ca argument al său
  - Trebuie inclus și un *constructor fără argumente* al cărui corp începe cu un apel la **super**
    - Acest apel la **super** trebuie să folosească *sirul implicit* ca argument



# Aruncarea unei excepții într-o metodă

- Uneori are sens să se arunce o excepție într-o metodă fără a o intercepta în metoda respectivă
  - Unele programe care folosesc o anume metodă ar trebui să se termine pur și simplu la aruncarea unei excepții, iar altele nu
  - În astfel de cazuri, programul care folosește invocarea metodei ar trebui să o includă într-un bloc **try** și să intercepteze excepția într-un bloc **catch** care urmează
- În acest caz, metoda în sine nu va include blocuri **try** și **catch**
  - Totuși, trebuie să conțină o clauză **throws**

Computer Science



# Declararea exceptiilor în clauza `throws`

- Dacă o metodă poate arunca o excepție, dar nu o interceptează, atunci ea trebuie să furnizeze un avertisment
  - Acest avertisment se numește *clauză throws*
  - Procesul de includere a unei clase excepție într-o clauză throws se numește *declararea excepției*  
`throws OExceptie //clauza throws`
  - Următorul cod declară că invocarea lui `oMetoda` poate cauza aruncarea lui `OExceptie`  
`public void oMetoda() throws OExceptie`
- `main` este o metodă care poate avea și ea specificarea unei excepții:

```
public static void main(String[] args) throws Exception
```



# Declararea exceptiilor în clauza `throws`

- Dacă o metodă poate arunca mai mult de un fel de excepție, atunci tipurile se separă prin virgule

```
public void oMetoda() throws OExceptie, AltaExceptie
```

- Dacă o metodă aruncă o excepție și nu o interceptează, atunci apelul metodei se termină imediat



# Regula "prinde sau declară"

- Cele mai obișnuite excepții care ar putea fi aruncate într-o metodă trebuie tratate în unul dintre următoarele două moduri:
  1. Codul care poate arunca o excepție este pus într-un bloc **try**, iar excepția care poate apărea este interceptată într-un bloc **catch** din aceeași metodă
  2. Excepția posibilă poate fi declarată la începutul definiției metodei punând numele clasei excepție într-o clauză **throws**

Computer Science



# Regula "prinde sau declară"

- Prima dintre tehnici tratează o excepție într-un bloc **catch**
- Cea de a doua tehnică este o modalitate de a deplasa răspunderea pentru tratarea excepției la metoda care a invocat-o pe cea care a aruncat excepția
- Metoda apelantă trebuie să trateze excepția, cu excepția cazului în care folosește aceeași tehnică de "pasare"
- Într-un sfârșit, fiecare excepție ar trebui interceptată de un bloc **catch** din vreo metodă care nu numai declară într-o clauză **throws** ci și interceptează clasa de excepție respectivă



# Regula "prinde sau declară"

- În oricare metodă, ambele tehnici pot fi amestecate
  - Unele excepții pot fi interceptate, iar altele declarate în clauza **throws**
- Cu toate acestea, tehnicile menționate trebuie folosite consistent pentru o excepție dată
  - Dacă o excepție nu este declarată, atunci ea trebuie tratată în metodă
  - Dacă este declarată excepția, atunci responsabilitatea pentru tratarea ei este pasată unei alte metode care o apelează
  - Observați că dacă definiția unei metode include invocarea unei a doua metode, iar cea de a doua poate arunca o excepție și nu o interceptează, atunci prima metodă trebuie să o declare sau să o intercepteze



# Exceptii verificate și neverificate

- Exceptiile care sunt supuse regulii "prinde sau declară" sunt numite **exceptii verificate**
  - Compilatorul verifică pentru a vedea dacă exceptiile sunt luate în considerare fie într-un bloc **catch**, fie într-o clauză **throws**
  - Clasele **Throwable**, **Exception**, precum și toți descendenții clasei **Exception** (exceptie **RuntimeException**) constituie exceptii verificate
- **Toate celelalte exceptii sunt neverificate**
  - Clasele **Error** și **RuntimeException** și toate clasele care descind din ele constituie exceptii *neverificate*
  - Aceste clase *nu sunt* supuse regulii "prinde sau declară"

Computer Science



# Exceptii de la regula "prinde sau declară"

- Exceptiile *verificate* trebuie să respecte regula "prinde sau declară"
  - Programele în care pot fi aruncate aceste exceptii nu se vor compila până când exceptiile respective nu sunt tratate corespunzător
- Exceptiile *neverificate* nu sunt supuse regulii "prinde sau declară"
  - Programele în care apar astfel de exceptii trebuie pur și simplu corectate întrucât au erori de alt fel (dacă compilatorul semnalează erori)

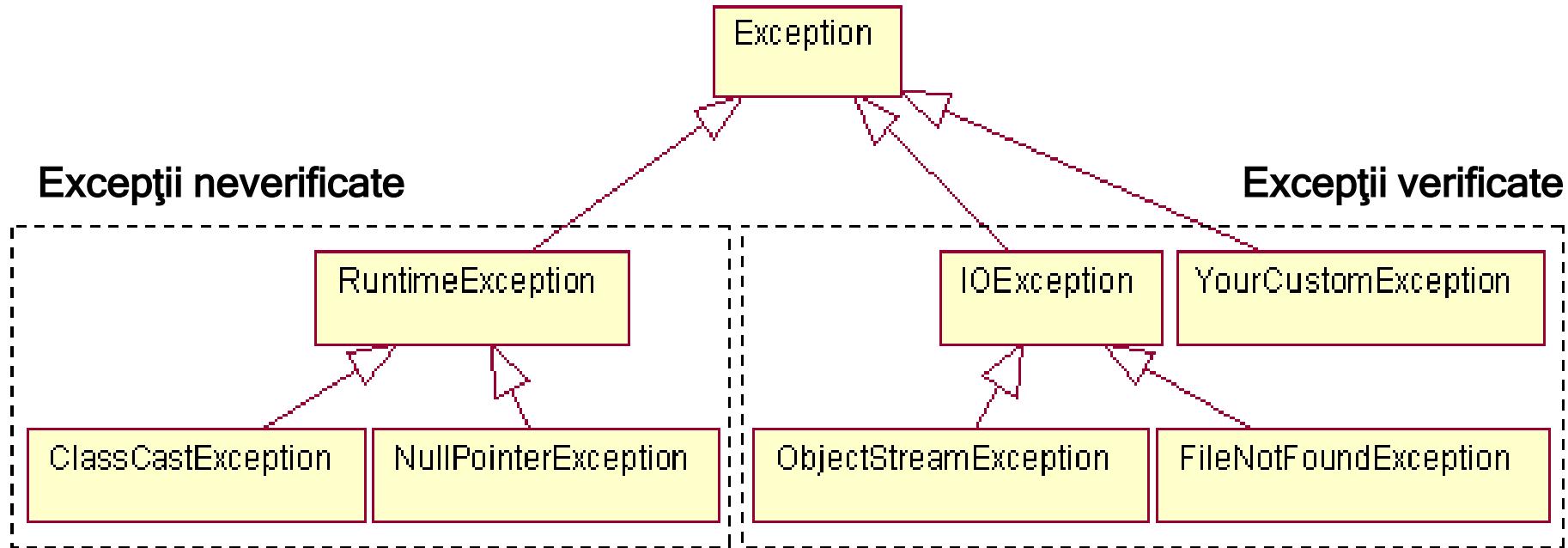
Computer Science



# Excepții verificate și neverificate

Excepții neverificate

Excepții verificate

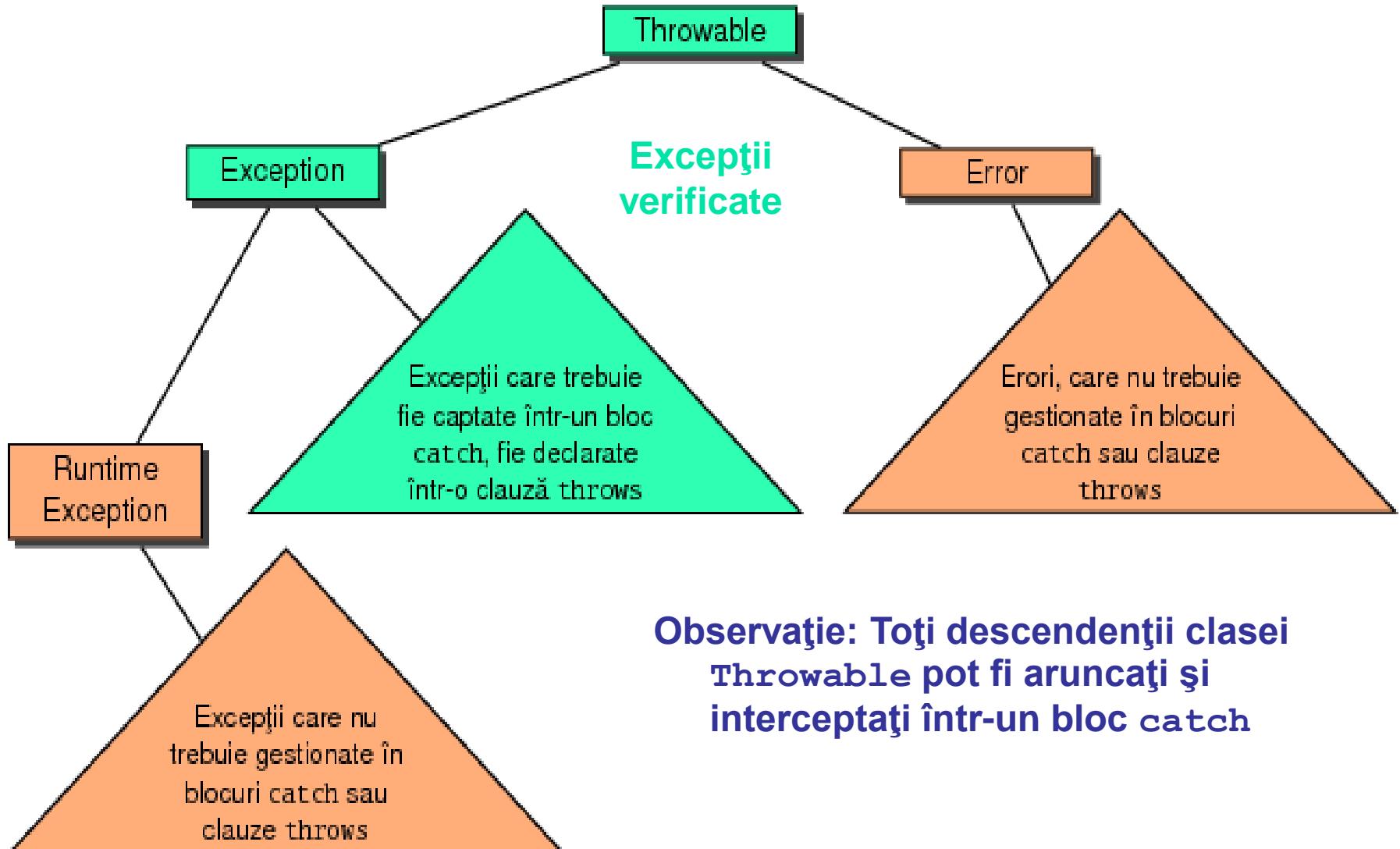


Notă. Aici este o mică parte a ierarhiei!



# Ierarhia obiectelor Throwable (aruncabile)

TECHNICAL UNIVERSITY



**Observație: Toți descendenții clasei `Throwable` pot fi aruncați și interceptați într-un bloc `catch`**



# Clauza **throws** în clase derivate

- La suprascrierea unei metode într-o clasa derivată, aceasta trebuie să aibă aceleași clase excepție precum cele listate în clauza **throws** din clasa de bază
  - sau un subset al acestora
- O clasă derivată *nu poate adăuga* excepții la clauza **throws**
  - dar poate șterge câteva

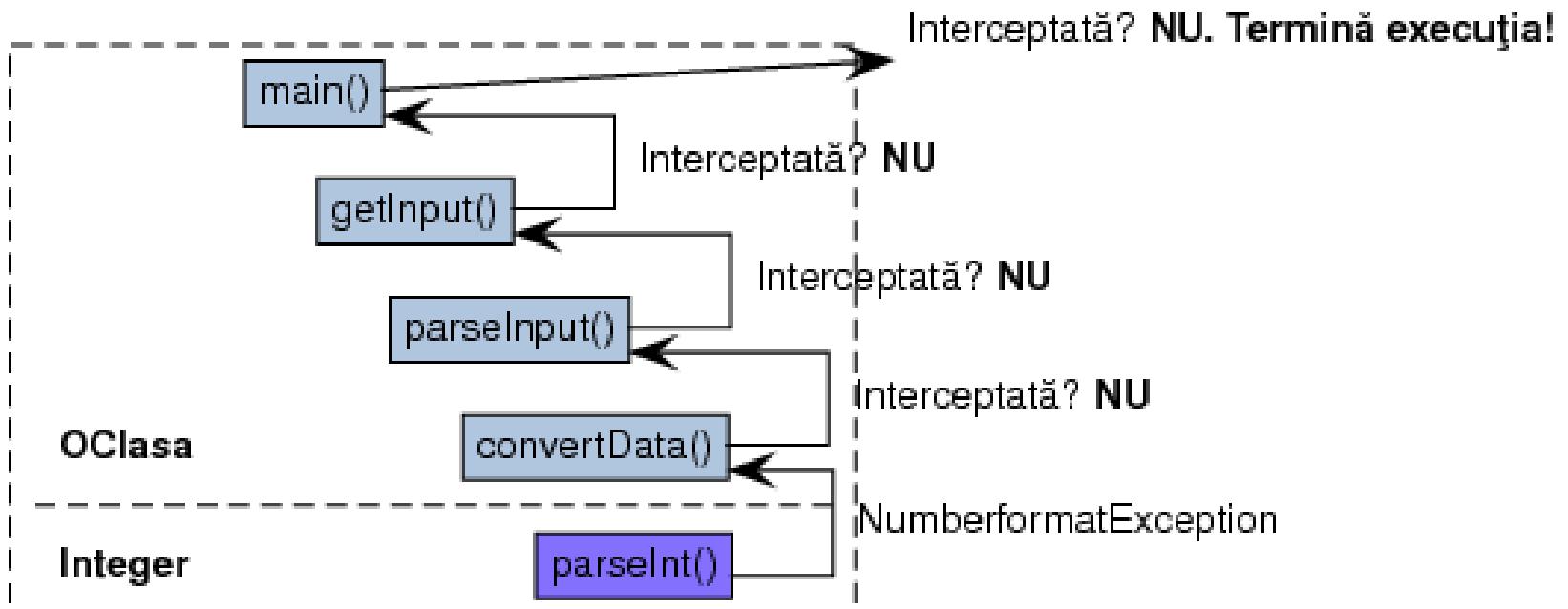


# Ce se întâmplă dacă o excepție nu este interceptată?

- Dacă fiecare metodă până la, și inclusiv, metoda **main** include o clauză **throws**, excepția respectivă poate fi aruncată, dar poate să nu fie interceptată niciodată
  - Într-un program GUI (adică un program cu o interfață cu ferestre, grafică) nu se întâmplă nimic – atâtă doar că utilizatorul poate fi lăsat într-o situație ne-explicată, iar programul poate să nu mai fie sigur
  - În programe non-GUI, aceasta face ca programul să se termine cu un mesaj de eroare care dă numele clasei excepție
- Fiecare program bine scris trebuie în cele din urmă să intercepteze fiecare excepție printr-un bloc **catch** în una dintre metodele sale



# Propagarea excepției



Computer Science



# Un alt exemplu

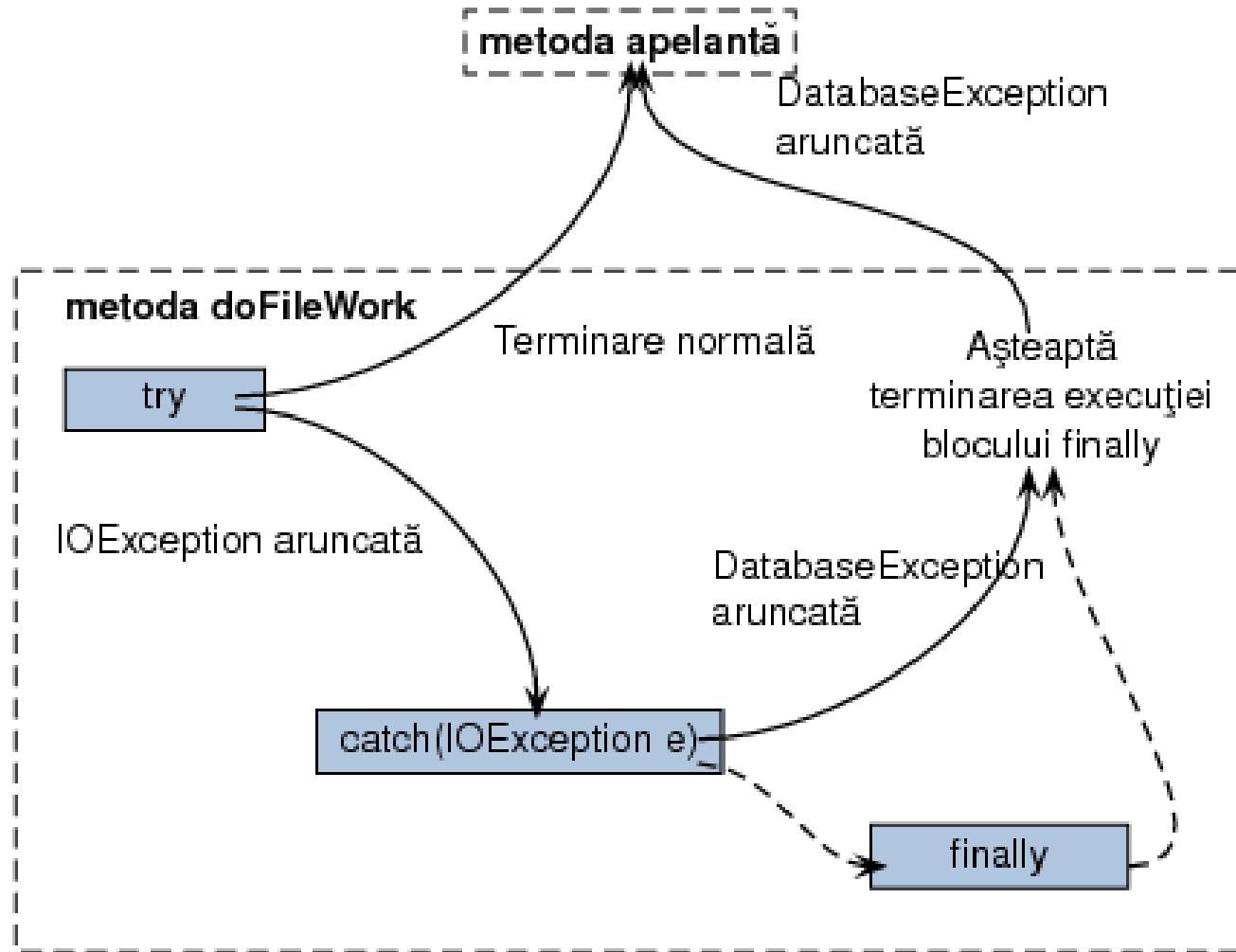
```
public void doFilework(String filename)
                        throws DatabaseException
{
    FileOutputStream fos = null;
    ObjectOutputStream oos = null;
    try{
        fos = new FileOutputStream(filename);
        oos = new ObjectOutputStream(fos);
        oos.writeObject(obj);
    }
    catch(IOException e){
        throw new DatabaseException(
            "Problem while working with "
            +filename+": "+ e.getMessage());
    }
}
```

```
finally{
    try{
        if(oos!=null){
            oos.close();
        }
        if(fos!=null){
            fos.close();
        }
    }
    catch(IOException e){
        throw new DatabaseException(
            "Problem while working with "
            +filename+": "+e.getMessage());
    }
}
```

Computer Science



# Explicații pentru exemplu





# Când să folosim excepțiile

- Excepțiile trebuie rezervate pentru situațiile în care o metodă întâlnește *un caz neobișnuit sau neașteptat, care nu poate fi tratat cu ușurință în vreun alt mod*
- Atunci când trebuie folosită tratarea excepțiilor, folosiți aceste recomandări:
  - Includeți instrucțiuni **throw** și precizați clasele excepție într-o clauză **throws** din definiția metodei respective
  - Plasați blocurile **try** și **catch** într-o metodă diferită

Computer Science



# Când să folosim excepțiile

- Iată un exemplu de metodă din care este aruncată o excepție:

```
public void oMetoda() throws OExceptie
{
    . . .
    throw new OExceptie(UnArgument);
    . . .
}
```

- Atunci când **oMetoda** este folosită de **altaMetoda**, **altaMetoda** trebuie să trateze excepția:

```
public void altaMetoda()
{
    try {
        oMetoda();
        . . .
    }
    catch (OExceptie e) {
        CodPentruTratareaExcepiei
    }
    . . .
}
```



# Ghid pentru exceptii

- Dacă metoda întâlnește o condiție anormală pe care nu o poate trata, atunci trebuie să arunce o excepție
- Evitați folosirea excepțiilor pentru a indica situații care pot fi așteptate ca parte a funcționării normale a metodei
- Dacă metoda descoperă că clientul și-a încălcă obligațiile contractuale (spre exemplu, prin transmiterea de date de intrare neconforme specificației), atunci aruncați o excepție neverificată

Computer Science



# Ghid pentru exceptii

- Dacă metoda nu-și poate îndeplini contractul, atunci aruncați fie o excepție verificată, fie una neverificată
- Dacă aruncați o excepție pentru o situație anormală despre care considerați că programatorii trebuie să decidă în mod conștient cum să o trateze, atunci aruncați o excepție verificată
- Definiți sau alegeti o clasă excepție care există deja pentru fiecare fel de condiție anormală care poate face ca metoda dvs. să arunce o excepție



# Re-aruncarea excepțiilor

- După interceptarea unei excepții, ea poate fi re-aruncată dacă e cazul
- La re-aruncarea unei excepții putem alege locația din care se va vedea ca aruncat obiectul în trasarea stivei de execuție
  - Putem face ca excepția re-aruncată să pară a fi aruncată din locul excepției originale sau din locul re-aruncării
- Pentru a re-arunca o excepție cu indicarea locației originale, pur și simplu o aruncăm din nou:

```
try {  
    cap(0);  
}  
  
catch(ArithmeticException e) {  
    throw e;  
}
```



# Re-aruncarea exceptiilor

- Pentru a avea locația reală din care a fost re-aruncată apelăm metoda **fillInStackTrace()** a exceptiei
  - Metoda setează informația din trasarea stivei pe baza contextului de execuție curent. Exemplu:

```
try {  
    cap(0);  
}  
catch(ArithmetricException e) {  
    throw (ArithmetricException)e.fillInStackTrace();  
}
```
- Apelăm **fillInStackTrace()** pe linia cu instrucțiunea **throw** – astfel numărul de linie din trasare este la fel cu cel unde apare instrucțiunea **throw**
  - Metoda **fillInStackTrace()** returnează o referință la clasa **Throwable**, aşa că e nevoie de o conversie de tip la tipul real de exceptie



# Exemplu de apel al metodei **fillInStackTrace()**

```
import java.lang.*;  
  
public class ThrowableDemo {  
  
    public static void function1() throws Exception {  
        throw new Exception("this is thrown from  
                           function1()");  
    }  
  
    public static void function2() throws Throwable{  
        try {  
            function1();  
        }  
        catch(Exception e) {  
            System.err.println("Inside function2():");  
            e.printStackTrace();  
            throw e.fillInStackTrace();  
        }  
    }  
}
```

```
public static void main(String[]  
                      args) throws Throwable {  
    try {  
        function2();  
    }  
    catch (Exception e) {  
        System.err.println("Caught  
                           Inside Main:");  
        e.printStackTrace();  
    }  
}
```



# Exemplu de apel al metodei **fillInStackTrace()**

- Rezultatul execuției acestui exemplu este:

Inside function2():

```
java.lang.Exception: this is thrown from function1()
at ThrowableDemo.function1(ThrowableDemo.java:4)
at ThrowableDemo.function2(ThrowableDemo.java:9)
at ThrowableDemo.main(ThrowableDemo.java:19)
```

Caught Inside Main:

```
java.lang.Exception: this is thrown from function1()
at ThrowableDemo.function2(ThrowableDemo.java:13)
at ThrowableDemo.main(ThrowableDemo.java:19)
```



# Programare orientată pe obiecte

## 1. Colecțiile Java



# Limitările tablourilor

- Tablourile nu sunt întotdeauna cea mai bună soluție pentru gestiunea seriilor, seturilor și a grupurilor de date
- Tablourile nu excelează atunci când mărimea setului de date poate fluctua
  - Inserarea unui element necesită glisarea elementelor de deasupra punctului de inserție -> e nevoie de spațiu suplimentar alocat la sfârșit
- Tablourile expun programatorului de aplicație problemele legate de gestiunea de nivel jos a memoriei
- Dacă cineva dorește să ofere accesul la un tablou privat
  - Poate face accesibil tabloul însuși printr-o metodă accesașe
  - Poate furniza o interfață pentru iterare cu metodele: first, next, etc.
  - Poate întoarce o copie adâncă a tabloului – lucru foarte ineficient



# Colecții versus tablouri

- Lucrul cu colecții (*API Collection*) este diferit de lucrul cu tablouri
- Tablourile sunt cu *legare tare la tipuri (strongly typed)*
  - Se specifică tipul elementelor și compilatorul impune tipul la încercarea de asignare de valori la elemente
  - Se pot defini tablouri de elemente de tipuri primitive
- Colectiile sunt cu *legare slabă la tipuri (weakly typed)*
  - Există o clasă **Vector** și toate elementele sale sunt de tipul **Object** -> toate obiectele de toate tipurile pot fi stocate acolo și e nevoie de forțarea tipului (downcast) elementelor la citire
  - Nu se pot include valori primitive în colecții, deși există o cale de împachetare/învelire (box) a lor – clasele învelitoare (d.e. Integer, Boolean, Double etc.)



# Colecții versus tablouri

- Tablourile dau în general viteză de execuție mai mare, deoarece reprezintă blocuri de memorie accesibile direct
- Colecțiile sunt obiecte cu metode care trebuie invocate pentru a citi sau scrie elemente
- Colecțiile sunt mult mai ușor de folosit pentru programare de uz general, în special atunci când datele sunt foarte volatile
  - Multe adăugări, ștergeri și modificări directe în timp



# Colecții

- **Colecție Java:** orice clasă care păstrează obiecte și implementează interfața **Collection**
  - De exemplu, clasa **ArrayList<T>** este o clasă colecție Java și implementează toate metodele din interfața **Collection**
  - Colecțiile sunt folosite împreună cu *iteratori*
- Interfața **Collection** este cel mai înalt nivel din cadrul general Java pentru clase colecție
  - Toate clasele colecție tratate aici se află în pachetul **java.util**

Computer Science



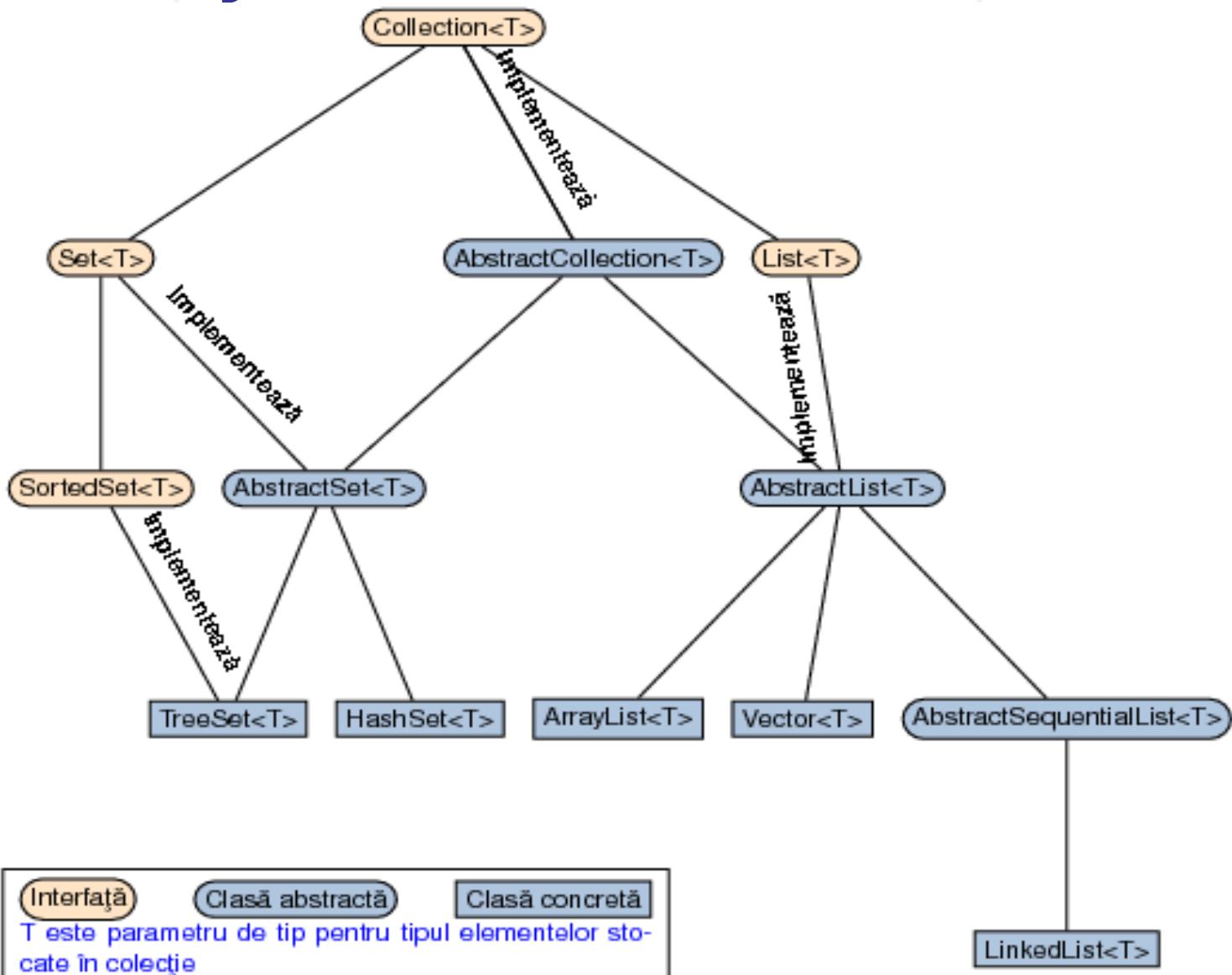
# Colecții

- *API Collection* include:

- Colecții cum sunt **Vector**, **LinkedList** și **Stack**
- *Mapări* care indexează valori pe baza cheilor, cum este **HashMap**
- Variante care asigură că elementele sunt întotdeauna ordonate de un comparator: **TreeSet** și **TreeMap**
- *Iteratori* care abstractizează abilitatea de a citi și scrie conținutul colecțiilor în bucle și care izolează acea abilitate de implementarea colecției care stă la bază



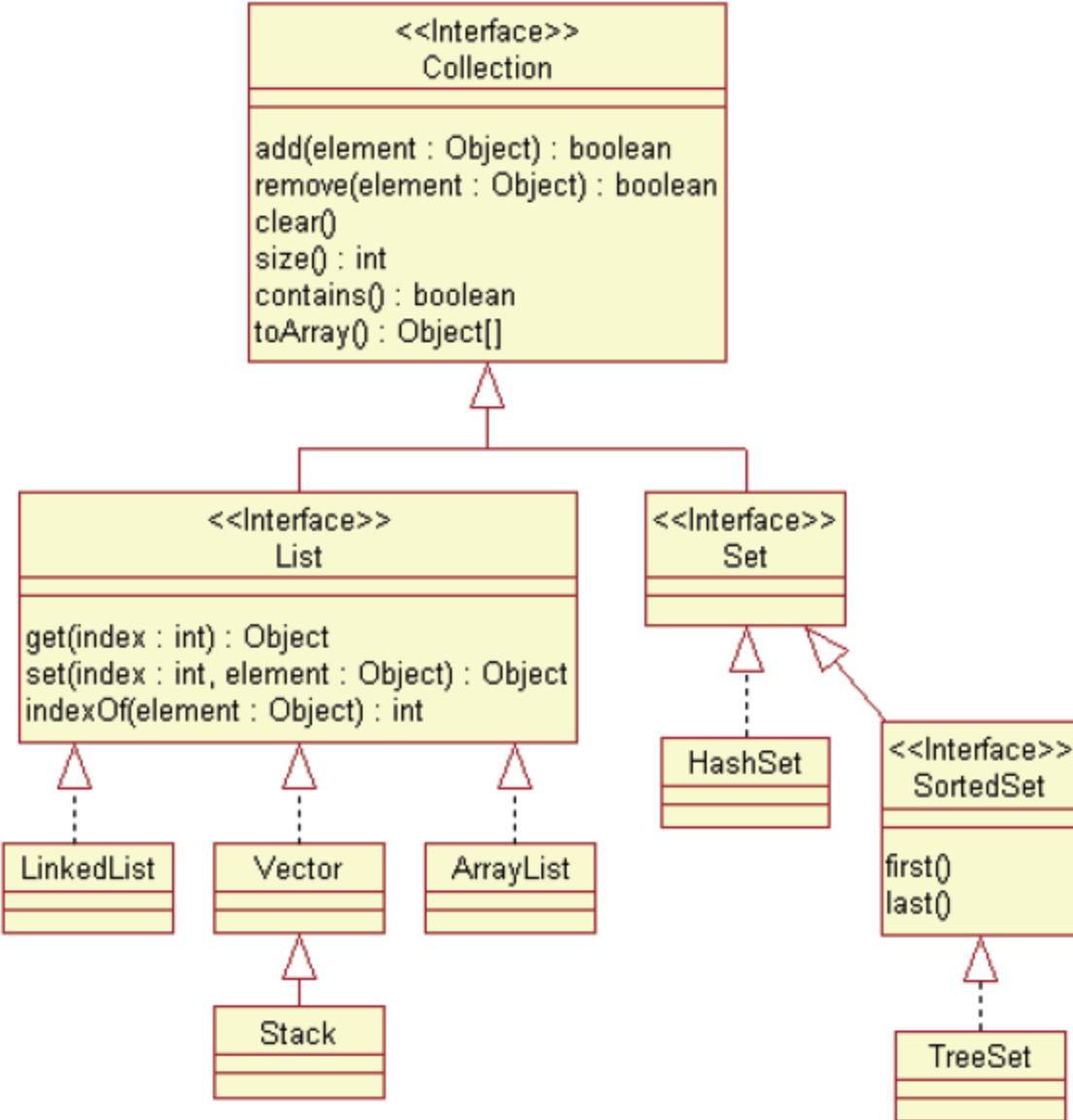
# "Peisajul" Collection





# "Peisajul" Collection

- Colectiile ordonate implementeaza **List**
- Colectiile care asigură unicitatea elementului implementează **Set**
- Colectiile sortate implementează **SortedSet**





# Caractere de nume nespecificat (wildcards)

- Clasele și interfețele din cadrul general **Collection** pot avea specificări de parametri de tip care nu specifică complet tipul care îl va avea parametrul
  - Pentru că ele specifică o gamă largă de tipuri de argumente, ele sunt numite **caractere de nume nespecificat** (*wildcards*)

```
public void method(String arg1, ArrayList<?> arg2)
```

- În exemplul de mai sus, primul argument este de tipul **String**, în timp ce al doilea argument poate fi un **ArrayList<T>** cu orice tip de bază

Computer Science



# Caractere de nume nespecificat (wildcards)

- Se poate limita efectul unui caracter de nume nespecificat (wildcard) precizând că tipul trebuie să fie un tip strămoș sau descendant al unei clase sau a unei interfețe
  - Notația `<? extends String>` specifică faptul că argumentul care va fi folosit trebuie să fie un obiect din orice clasă descendenta a lui `String`
  - Notația `<? super String>` specifică faptul că argumentul care va fi folosit trebuie să fie un obiect din orice clasă strămoș a lui `String`



# Cadrul general Collection

- Interfața **Collection<T>** descrie operațiile de bază pe care toate clasele colecție trebuie să le implementeze
- Cum o interfață este un tip, orice metodă poate avea parametri de tipul **Collection<T>**
  - Parametrul respectiv poate fi înlocuit la apel cu orice argument care este un obiect de orice clasă din cadrul general colecție



# Interfața Collection

- Toate colecțiile pot:
  - adăuga / elimina elemente
  - șterge toate elementele colecției astfel încât rezultă un set vid
  - raporta mărimea lor
  - converti datele într-un tablou de **Object**
- Se pot defini proprietăți suplimentare definite de implementarea uneia dintre sub-interfețele **Collection**
  - colecțiile ordonate implementează **List**
  - colecțiile care asigură unicitatea elementelor implementează **Set**
  - colecțiile care sortează implementează **SortedSet**

```
interface Collection
{ // lista parțială de metode
  public int size();
  public void clear();
  public Object[] toArray();
  public boolean add( Object );
  public boolean remove( Object );
  public boolean addAll( Collection );
  public Iterator iterator()
```



# Construirea colecțiilor

- Colecțiile trebuie *create* explicit
  - Greșală frecventă: declararea unei referințe la un **Vector** sau la o **LinkedList** și presupunerea că obiectul este acolo
- Odată creat obiectul colecție, pur și simplu îl se adaugă elemente
  - Folosiți **add** pentru a adăuga un nou element la sfârșit. Atenție că valorile primitive trebuie "învelite" în clase:

```
Vector<Integer> vec = new Vector<Integer>();  
vec.add(new Integer(5));  
vec.add(new Integer(4));  
System.out.println(vec); // Output: [5, 4]  
int i = ((Integer) vec.elementAt(0)).intValue();
```
  - Folosiți metodele de inserare – definite de subtipuri ale **Collection**
- Folosiți **remove** pentru a elimina un element, identificându-l
  - Multe subtipuri oferă metode de eliminare bazate pe indecsi
- Se poate pune orice obiect Java în orice colecție
  - Colecții omogene vs eterogene



# Clase colecție concrete

- Clasele `ArrayList<T>` și `Vector<T>` implementează toate metodele din interfața `List<T>` și pot fi folosite aşa cum sunt dacă nu e nevoie de metode suplimentare
  - Fiecare dintre ele se poate folosi atunci când este nevoie de o `List<T>` cu acces aleatoriu eficient la elemente
- Clasa concretă `HashSet<T>` implementează toate metodele din interfața `Set<T>` și poate fi folosită aşa cum este dacă nu e nevoie de metode suplimentare
  - `HashSet<T>` adaugă doar constructori pe lângă metodele din interfață
  - `HashSet<T>` este implementată folosind o *tabelă de dispersie*

Computer Science



# Clasa Vector

- Oferă accesul aleatoriu la o listă scalară de elemente
  - **Vector** și **ArrayList** au semantica apropiată de un tablou:

```
for (int n = 0; n < vec.size(); n++)
    System.out.println((String) vec.elementAt(n));
```
  - Elementele sunt în ordinea în care au fost adăugate la colecție – nu există sortare implicită
  - Elementele nu trebuie să fie unice în colecție
- Vectorii se comportă cel mai bine la "acces aleator" la elemente – au în spate tablouri
  - punctul slab – ca și la tablouri – inserarea și ștergerea
- Vectorii au capacitate și mărime
  - **size** = mărimea; numărul de elemente aflate curent în colecție
  - capacitate = este dimensiunea tabloului **Object[] elementData** din clasa Vector; capacitatea se incrementează automat cu **capacityIncrement** de fiecare dată cand **size** devine mai mare decât capacitatea
  - capacitate  $\geq$  **size**



# Clasa ArrayList

- Crearea:
  - `new ArrayList()`
  - `new ArrayList(int initialCapacity)`
- Măsurarea:
  - `int size()`
- Stocarea:
  - `boolean add(Object o)`
  - `boolean add(int index, Object element)`
  - `Object set(int index, Object element)`
- Regăsirea:
  - `Object get(int index)`
  - `Object remove(int index)`
- Testarea:
  - `boolean isEmpty()`
  - `Boolean contains(Object elem)`
- Aflarea poziției (eșec = -1):
  - `int indexOf(Object elem)`
  - `int lastIndexOf(Object elem)`



# Diferențe între `ArrayList<T>` și `Vector<T>`

- Pentru majoritatea scopurilor, `ArrayList<T>` și `Vector<T>` sunt echivalente
  - Clasa `Vector<T>` este mai veche și a trebuit să i se adauge câteva metode pentru a se potrivi în cadrul general colecție
  - Clasa `ArrayList<T>` este mai nouă și a fost creată ca parte a cadrului general colecție Java
  - Clasa `ArrayList<T>` se presupune a fi și mai eficientă decât clasa `Vector<T>`



# Exemplu ArrayList

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Date;

public class IteratorReferenceDemo
{
    public static void main(String[] args)
    {
        ArrayList<Date> birthdays =
            new ArrayList<Date>();
        birthdays.add(new Date(90, 1, 1));
        birthdays.add(new Date(90, 2, 2));
        birthdays.add(new Date(90, 3, 3));
        System.out.println("Lista contine:");
        Iterator<Date> i =
        birthdays.iterator();
        while (i.hasNext())
            System.out.println(i.next());
        i = birthdays.iterator();
        Date d = null;
        System.out.println("Schimbarea
                           referintelor.");
    }
}
```

```
while (i.hasNext( ))
    d = i.next( );
    d.setDate(1);
    d.setMonth(3);
    d.setYear(90);
}
System.out.println("Lista contine
acum:");
i = birthdays.iterator();
while (i.hasNext( ))
    System.out.println(i.next());
}
}
```

## Rezultate afisate:

Lista contine:  
Thu Feb 01 00:00:00 EET 1990  
Fri Mar 02 00:00:00 EET 1990  
Tue Apr 03 00:00:00 EEST 1990  
Schimbarea referintelor.  
Lista contine acum:  
Sun Apr 01 00:00:00 EEST 1990  
Sun Apr 01 00:00:00 EEST 1990  
Sun Apr 01 00:00:00 EEST 1990



# Exemplu: Aflarea sirurilor dupicate

```
import java.util.*;
public class FindDups {
    public static void main(String args[]) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            if (!s.add(a))
                System.out.println("Duplicat: " + a);
        System.out.println(s.size() +
                           " cuvinte distincte: " + s);
    }
}
```

## Rulare:

```
java FindDups i came i saw i learned
```

## Rezultate afisate:

Duplicat: i

Duplicat: i

4 cuvinte distincte: [i, learned, saw, came]

- Remarcați faptul că **codul referă întotdeauna Colectia prin tipul interfeței sale (Set)** nu prin tipul implementării (**HashSet**)
- Aceasta este o **practică de programare foarte recomandată** deoarece vă oferă flexibilitatea de a schimba implementările prin simpla schimbare a constructorului



# Exemplu modificat : Aflarea sirurilor dupicat

```
import java.util.*;
public class FindDups2 {
    public static void main(String args[]) {
        Set<String> uniques = new HashSet<String>();
        Set<String> dups = new HashSet<String>();
        for (String a : args)
            if (!uniques.add(a)) dups.add(a);

        // Diferenta de multimi distractiva
        uniques.removeAll(dups);
        System.out.println("Cuvinte unice: " + uniques);
        System.out.println("Cuvinte duplicate: " + dups);
    }
}
```

## Rulare:

```
java FindDups i came i saw i learned
```

## Rezultate afisate:

```
Cuvinte unice: [learned, saw, came]
```

```
Cuvinte duplicate: [i]
```



# Clase colecție concrete

- Clasa concretă **LinkedList<T>** este derivată din clasa abstractă **AbstractSequentialList<T>**
  - Ar trebui folosită atunci când este nevoie de traversarea secvențială eficientă a unei liste
- Interfața **SortedSet<T>** și clasa concretă **TreeSet<T>** sunt destinate să implementeze interfața **Set<T>** și să ofere regăsirea rapidă a elementelor
  - Implementarea clasei este asemănătoare cu un arbore binar, dar inserarea păstrează echilibrul arborelui

Computer Science



# Clasa `LinkedList`

- Este un alt mijloc de obținere a unei colecții scalare
  - Fiecare element din lista înlănuțuită este discret în memorie
  - Elementul conține o referință spre elementul următor și alta spre elementul precedent
- Listele înlănuțuite se comportă bine la inserări și ștergeri – nu este nevoie de glisarea elementelor la inserare
  - Se desfac legăturile existente și se formează altele noi
  - Ștergerea implică schimbarea unor legături
- Iterarea este mai lentă
  - Nu se poate căuta aleator, trebuie traversată element cu element

Computer Science



# O privire asupra cadrului general Map

- Cadrul general Java *map* tratează colecții de *perechi ordonate*
  - De exemplu, o cheie și valoarea asociată ei
- Obiectele din cadrul general *map* pot implementa funcții și relații, astfel încât pot fi folosite la construirea claselor pentru baze de date
- Cadrul general *map* folosește interfața **Map<T>**, clasa **AbstractMap<T>** și clase derivate din aceasta

Computer Science



# Enumerări și iteratori

- Sunt obiecte folosite pentru a parcurge un container
  - Sunt disponibile pentru unele clase container standard – care implementează interfețele corespunzătoare
  - Funcționează corect chiar dacă containerul se modifică
  - Ordinea poate sau nu să fie semnificativă
- Java are două variațiuni:
  - **Enumeration** (vechi: de la JDK 1.0)
  - **Iterator** (mai nou: de la JDK 1.2)



# Enumerări

TECHNICAL UNIVERSITY

- Pentru a obține un enumerator **e** pentru containerul **v**:
  - **Enumeration e = v.elements();**
  - **e** este inițializat la începutul listei
- Pentru a obține primul element și următoarele:
  - **someObject = e.nextElement()**
- Pentru a verifica dacă le-am parcurs pe toate:
  - **e.hasMoreElements()**
- Exemplu:

```
for (Enumeration e=v.elements(); e.hasMoreElements();)  
{  
    System.out.println(e.nextElement());  
}
```

Computer Science



# Iteratori

TECHNICAL UNIVERSITY

- **Iterator**: un obiect folosit la o colecție pentru a furniza accesul secvențial la elementele colecției
  - Acest acces permite examinarea și eventual modificarea elementelor
- Iteratorul impune o ordonare a elementelor colecției chiar dacă colecția însine nu impune o ordine asupra elementelor pe care le conține
  - În cazul în care colecția impune o ordonare asupra elementelor sale, iteratorul va folosi aceeași ordonare

Computer Science



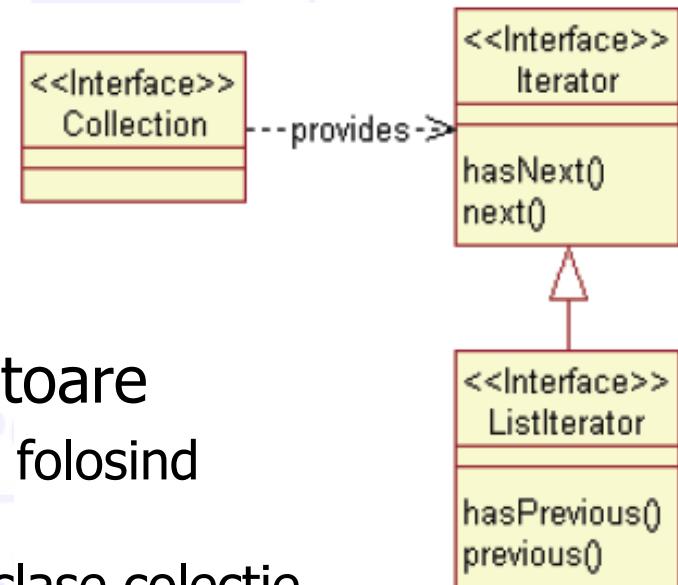
# Interfață Iterator<T>

- Interfața **Iterator<T>** izolează folosirea unei colecții de clasa colecție în sine

```
interface Iterator
{
    public boolean hasNext();
    public Object next();
    public void remove(); // optional
}
```

- **Iterator<T>** nu este de sine stătătoare
  - Ea trebuie asociată cu un obiect colecție folosind metoda **iterator**
  - Exemplu: dacă **c** este o instanță a unei clase colecție (d.e., **HashSet<String>**), codul care urmează obține un iterator pentru **c**:

```
Iterator iteratorForC = c.iterator();
```





# Folosirea unui iterator cu un obiect HashSet<T>

- Un obiect de tipul `HashSet<T>` nu impune nici o ordine asupra elementelor pe care le conține
- Cu toate acestea, un iterator va impune o ordine asupra elementelor din set
  - Aceasta va fi ordinea în care elementele sunt regăsite de `next()`
  - Deși la fiecare rulare a programului ordinea elementelor produse astfel poate fi identică, nu există nici o cerință care să impună acest lucru



# Exemplu: iterator peste HashSet<T>

```
import java.util.HashSet;
import java.util.Iterator;
public class HashSetIteratorDemo
{
    public static void main(String[] args)
    {
        HashSet<String> s = new HashSet<String>();
        s.add("health");
        s.add("love");
        s.add("money");
        System.out.println("The set contains:");
        Iterator<String> i = s.iterator();
        while (i.hasNext())
            System.out.println(i.next());
        i.remove();
        System.out.println();
        System.out.println("The set now contains:");
        i = s.iterator();
        while (i.hasNext()) System.out.println(i.next());
        System.out.println("End of program.");
    }
}
```

## **Rezultate afisate:**

The set contains:  
love  
money  
health

The set now contains:  
love  
money  
End of program.



## Sugestie: Bucle "for-each" ca iteratori

- Deși nu este iterator, bucla for-each poate servi în același scop ca un iterator
  - Bucla for-each se poate folosi pentru a parcurge fiecare element al unei colecții
- Buclele for-each pot fi folosite la oricare colecție menționată
- Buclele **for** obișnuite nu pot parcurge elementele dintr-un obiect colecție
  - Spre deosebire de elementele din tablouri, elementele obiectelor colecție nu sunt în mod normal asociate cu indici
- Deși bucla **for** obișnuită nu poate parcurge elementele unei colecții, bucla **for** îmbunătățită poate parcurge elementele unei colecții



# Bucla "for each"

- Sintaxa generală a instrucțiunii **for**-each (pentru fiecare) folosită la o colecție este

```
for (TipColectie NumeVariabila:NumColectie)  
    Instructiune
```

- Linia **for**-each de mai sus trebuie citită ca "pentru fiecare **NumeVariabila** din **NumColectie** execută ceea ce urmează
  - Remarcați că **NumeVariabila** trebuie declarată în interiorul fiecărei bucle, nu înainte
  - Remarcați, de asemenea, că se folosește simbolul "două puncte" (:) după **NumeVariabila**

Computer Science



# Exemplu de buclă "for each" ca iterator

```
import java.util.HashSet;
import java.util.Iterator;
public class ForEachDemo {
    public static void main(String[] args) {
        HashSet<String> s = new HashSet<String>();
        s.add("health");
        s.add("love");
        s.add("money");
        System.out.println("The set contains:");
        String last = null;
        for (String e : s) {
            last = e;
            System.out.println(e);
        }
        s.remove(last);
        System.out.println();
        System.out.println("The set now contains:");
        for (String e : s) System.out.println(e);
        System.out.println("End of program.");
    }
}
```

## **Rezultate afisate:**

The set contains:  
love  
money  
health

The set now contains:  
love  
money  
End of program.



# Folosirea genericelor

- Un *tip generic* se definește în termenii unui alt tip pe care îl colectează sau asupra căruia acționează în vreun fel, folosind parantezele unghiulare (<>)

- Exemplu: un **ArrayList<Point>** este un tablou-listă de obiecte **Point** (din pachetul **java.awt**)

```
ArrayList<Point> someList = new ArrayList<Point>();
```

- Permite compilatorului să surprindă o eroare de genul:  
**somelist.add(new Dimension(10, 10));**
  - Unui obiect colecție de un anumit tip i se va furniza un iterator specific tipului respectiv

```
Iterator<Point> each = someList.iterator();
```

- Atunci nu mai este necesar să se forțeze conversia la tipul necesar pentru rezultatele metodelor accesoare, d.e.

```
while (each.hasNext())
    each.next().x = 11;
```



# Interfața `ListIterator<T>`

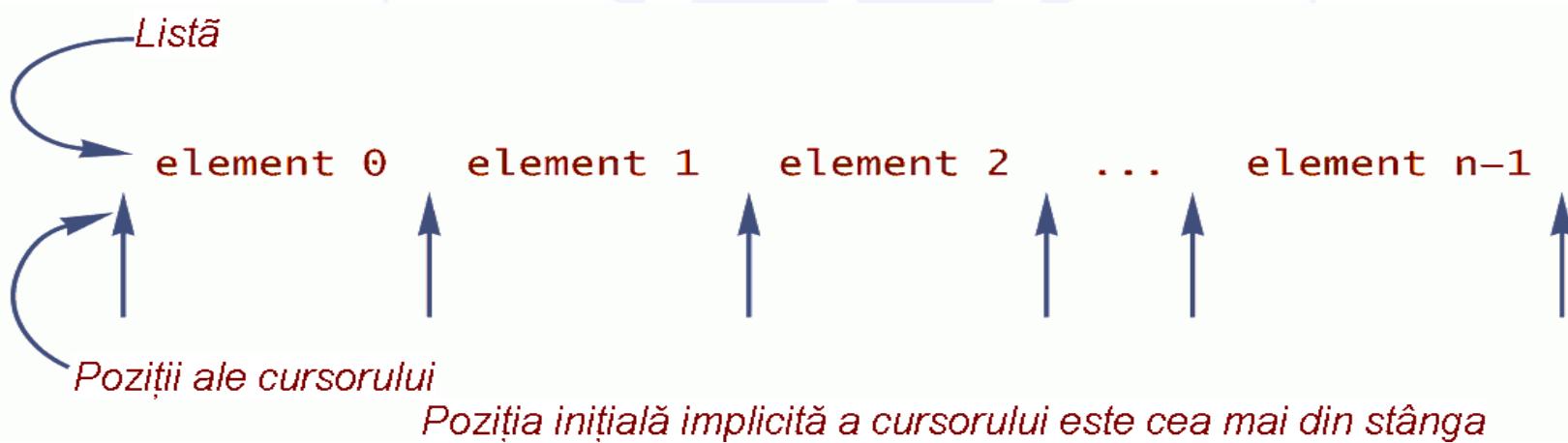
- Interfața `ListIterator<T>` extinde interfața `Iterator<T>` și este menită să lucreze cu colecții care satisfac interfața `List<T>`
  - Un `ListIterator<T>` are toate metodele pe care le are un `Iterator<T>`, plus metode suplimentare
  - Un `ListIterator<T>` se poate deplasa în ambele direcții pe o listă de elemente
  - `ListIterator<T>` are metode cum sunt `set()` și `add()` care se pot folosi la modificarea elementelor

Computer Science



# Cursorul ListIterator<T>

- Fiecare **ListIterator<T>** are un marcator de poziție numit *cursor*
  - Dacă lista are  $n$  elemente, atunci acestea sunt numerotate prin indici de la 0 la  $n-1$ , dar există  $n+1$  poziții ale cursorului
  - La apelul metodei **next()**, se returnează elementul care urmează imediat după cursor, iar cursorul este deplasat înainte cu o poziție
  - La invocarea metodei **previous()** se returnează elementul care urmează imediat înaintea cursorului, iar cursorul este deplasat înapoi cu o poziție de cursor





# Capcană: `next` și `previous` pot întoarce o referință

- Teoretic, atunci când o operație a iteratorului întoarce un element al colecției, el poate returna o copie sau o clonă a elementului sau poate returna o referință la element
- Iteratorii pentru clasele colecție standard predefinite, cum sunt `ArrayList<T>` și `HashSet<T>`, returnează de fapt referințe
  - De aceea, modificarea valorii returnate va face modificarea elementului din colecție



# Sugestie: Definirea Claselor Iterator proprii

- De obicei nu prea este nevoie de clase `Iterator<T>` sau `ListIterator<T>` definite de programator
- Cea mai simplă și mai utilizată cale pentru a *defini* o clasă colecție este *să o facem o clasă derivată* a uneia dintre clasele colecție de bibliotecă
  - Procedând astfel, metodele `iterator()` și `listIterator()` devin automat disponibile programului
- Dacă o clasă colecție trebuie definită în vreun alt mod, atunci clasa iterator ar trebui definită ca clasă internă (clasă imbricată) în clasa colecție



# Sortarea colecțiilor/tablourilor

- Folosind metoda statică **sort** a clasei **Collections** (colecții)
- Folosind metoda statică **sort** a clasei **Arrays** (tablouri)
- Sortarea unei colectii/tablou după criteriul definit de metoda **compareTo** a clasei din care fac parte obiectele
- Exemplu (sortare colecție):

```
import java.util.*;  
class TestSort {  
    public static void main(String[] args) {  
        ArrayList<String> stuff = new ArrayList<String>(); // #1  
        stuff.add("Denver");  
        stuff.add("Boulder");  
        stuff.add("Vail");  
        stuff.add("Aspen");  
        stuff.add("Telluride");  
        System.out.println("unsorted " + stuff);  
        Collections.sort(stuff); // #2  
        System.out.println("sorted " + stuff);  
    }  
}
```

Rezultate afișate:

```
unsorted [Denver, Boulder, Vail, Aspen, Telluride]  
sorted [Aspen, Boulder, Denver, Telluride, Vail]
```



# Sortarea colecțiilor/tablourilor

- Definirea mai multor criterii de sortare ale aceleiași colecții/aceluiași tablou folosind **comparatori**
  - Comparatorii sunt folosiți ca și argumente la ceva ce sortează
    - Metode de sortare
    - Structuri de date care sortează
  - Se utilizează interfața **java.util.Comparator**
    - Sunt create obiecte care sunt transmise metodelor de sortare sau structurilor de date care sortează
    - Un Comparator trebuie să definească o metodă **compare** care primește ca și argumente două Object și returnează un întreg <0, 0, sau >0 (mai mic, egal, mai mare) comparând primul Object cu cel de-al doilea



# Sortarea colecțiilor/tablourilor

## ■ Exemplu

- Listarea conținutului directorului implicit (home) al unui utilizator
- Demonstrează folosirea interfeței Comparator pentru a sorta același tablou după două criterii diferite
  - Directoarele înaintea fișierelor, apoi alfabetic
  - După lungimea numelui de fișier/director, cel mai lung primul



# Sortarea colecțiilor/tablourilor

```
// Author: Fred Swartz 2006-Aug-23 Public domain.

import java.util.Arrays;
import java.util.Comparator;
import java.io.*;

public class Filelistsort {

    //=====
    public static void main(String[] args) {
        //... Creaza comparatorii pentru sortare.
        Comparator<File> byDirThenAlpha = new DirAlphaComparator();
        Comparator<File> byNameLength = new NameLengthComparator();

        //... Creaza un obiect a File pentru directorul utilizatorului.
        File dir = new File(System.getProperty("user.home"));
        File[] children = dir.listFiles();
```



# Sortarea colecțiilor/tablourilor

```
System.out.println("Fisierele dupa director, apoi alfabetic ");
Arrays.sort(children, byDirThenAlpha);
printFileNames(children);

System.out.println("Fisierele dupa lungimea numelui lor
                  (cel mai lung primul)");
Arrays.sort(children, byNameLength);
printFileNames(children);
}

//=====
private static void printFileNames(File[] fa){
    for (File oneEntry : fa) {
        System.out.println(" " + oneEntry.getName());
    }
}
}
```



# Sortarea colecțiilor/tablourilor

```
//////////////////////////// DirAlphaComparator
// Pentru a sorta directoarele inaintea fisierelor, apoi alfabetic.
class DirAlphaComparator implements Comparator<File> {

    // Interfata Comparator necesita definirea metodei compare.
    public int compare(File filea, File fileb) {
        //... Sorteaza directoarele inaintea fisierelor,
        //      altfel alfabetic fara a tine seama de majuscule/minuscule.
        if (filea.isDirectory() && !fileb.isDirectory()) {
            return -1;

        } else if (!filea.isDirectory() && fileb.isDirectory()) {
            return 1;

        } else {
            return filea.getName().compareToIgnoreCase(fileb.getName());
        }
    }
}
```



# Sortarea colectiilor/tablourilor

```
///////////////////////////// NameLengthComparator
// Pentru a sorta dupa lungimea numelui de fisier/director
// (cel mai lung primul).
class NameLengthComparator implements Comparator<File> {

    // Interfata Comparator necesita definirea metodei compare.
    public int compare(File filea, File fileb) {
        int comp = fileb.getName().length() - filea.getName().length();
        if (comp != 0) {
            //... daca lungimile sunt diferite, am terminat.
            return comp;
        } else {
            //... daca lungimile sunt egale, sorteaza alfabetic.
            return filea.getName().compareToIgnoreCase(fileb.getName());
        }
    }
}
```

Computer Science



# Programare orientată pe obiecte

1. Testarea și depanarea programelor
2. Introducere în I/E Java



# Testarea

TECHNICAL UNIVERSITY

- *Testarea software*: procesul folosit la identificarea corectitudinii, completitudinii, securității și calității software
- *Testarea funcțională*: determină dacă sistemul satisface specificațiile clientului
- *Testarea tip cutie neagră*:
  - Proiectantul testelor ignoră structura internă a implementării
  - Testul este condus de comportamentul extern așteptat al sistemului
  - Sistemul este tratat ca o "cutie neagră": comportamentul este observabil, dar structura internă nu este cunoscută

Computer Science



# Proiectarea, planificarea și testarea cazurilor

- Proiectarea testelor începe de obicei cu analiza:
  - Specificațiilor funcționale ale sistemului
  - Cazurilor de utilizare: a modurilor în care va fi folosit sistemul
- Un caz de testare este definit de
  - Declararea obiectivelor cazului
  - Setul de date pentru caz
  - Rezultatele așteptate
- Un plan de teste este un set de cazuri de testare
- Pentru a dezvolta un plan de teste
  - Analizăm caracteristicile pentru a identifica cazurile de test
  - Considerăm seturile de stări posibile pe care le poate asuma un obiect
  - Testele trebuie să fie reprezentative



# Testarea unităților

- Cel mai important instrument de testare
- Verifică o singură metodă sau un set de metode care cooperează
- Nu testează întregul program în curs de dezvoltare; testează doar clasele luate izolat
- Pentru fiecare test furnizăm o clasă simplă numită *test harness* (engl. harness = ham, harnășament)
- *Test harness* alimentează cu parametri metodele care se testează

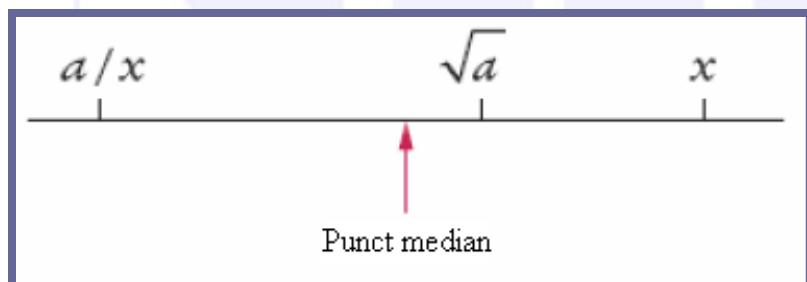
Computer Science



## Exemplu: Realizarea unui test

- Pentru a calcula rădăcina pătrată a lui  $a$  folosim un algoritm comun:

1. Ghicim o valoare a lui  $x$  care poate fi apropiată de rădăcină pătrată dorită ( $x = a$  este ok)
2. Rădăcina pătrată reală este undeva între  $x$  și  $a/x$
3. Luăm punctul median  $(x + a/x) / 2$  ca valoare mai bună pentru  $x$



4. Repetăm procedura (pasul 3) și ne oprim atunci când două valori succesive sunt foarte apropiate una de alta
- Metoda converge repede



# Testarea programului

- Clasa scrisă pentru rezolvarea problemei funcționează corect pentru toate intrările?
  - Trebuie testată cu mai multe valori
  - Re-testarea cu alte valori, în mod repetat, nu este o idee bună; testele nu sunt repetabile
  - Dacă se rezolvă o problemă și e nevoie de re-testare, e nevoie să ne reamintim toate intrările
- Soluție: scriem *teste harness* care să ușureze repetarea testelor de unități

Computer Science



# Furnizarea intrărilor pentru teste

- Există diverse mecanisme pentru furnizarea cazurilor de test
- Unul dintre acestea este scrierea intrărilor de test în codul *test harness* ("hardwire")
  - Pur și simplu se execută *test harness* ori de câte ori se rezolvă o eroare (*bug*) în clasa care se testează
  - Alternativă: să punem intrările într-un fișier
- Putem genera automat cazurile de testat
  - Pentru puține intrări posibile este fezabil să rulăm un număr (reprezentativ) de teste într-un ciclu
  - Testul anterior este restricționat la un subset mic de valori
  - Alternativa: generarea aleatoare a cazurilor de test



# Furnizarea intrărilor pentru teste

- Alegerea corespunzătoare a cazurilor de test este importantă în depanarea programelor
  - Testăm toate caracteristicile metodelor de testat
  - Testăm *cazurile tipice* – exemplu: 100, 1/4, 0.01, 2, 10E12, pentru problema descrisă anterior
  - Testăm *cazurile limită*: testăm cazurile care sunt la limita intrărilor acceptabile – exemplu: 0, pentru problema descrisă anterior
- Programatorii greșesc adesea la tratarea condițiilor limită
  - Împărțirea cu zero, extragerea de caractere din siruri vide, accesarea referințelor nule
- Adunăm cazuri de test negative: intrări pe care ne aşteptăm ca programul să le respingă
  - Exemplu: radical din -2, când testul trece dacă *harness* se termină cu eşecul aserţiunii (dacă este activată verificarea aserţiunilor)



# Citirea intrărilor dintr-un fișier

- E mai elegant să punem intrările pentru teste într-un fișier
- Redirectarea intrării: `java Program < data.txt`
- Unele IDE-uri nu suportă redirectarea intrării: în acest caz folosim fereastra de comandă (shell)
- Redirectarea ieșirii: `java Program > output.txt`
- Exemplu:
  - Fișierul test.in: 

```
100
4
2
1
0.25
0.01
```
  - Rularea programului:  
`java RootApproximatorHarness < test.in > test.out`



# Evaluarea cazurilor de test

- De unde știm dacă ieșirea este corectă?
- Calculăm valorile corecte cu mâna
  - D.e., pentru un program de salarizare, calculăm manual taxele
- Furnizăm intrări de test pentru care știm răspunsurile
  - D.e., rădăcina pătrată a lui 4 este 2, iar pentru 100 este 10
- Verificăm că valorile de ieșire satisfac anumite proprietăți
  - D.e., pătratul rădăcinii pătrate = valoarea inițială
- Folosim o altă metodă sigură pentru a calcula rezultatul în scop de testare
  - D.e., folosim `Math.pow` pentru a calcula mai lent  $x^{1/2}$  (echivalentul rădăcinii pătrate a lui  $x$ )



# Testarea regresivă

- Salvăm cazurile de test
- Folosim cazurile de test salvate în versiunile următoare
- *Suită de teste* : un set de teste pentru testarea repetată
- *Ciclarea* : eroare care a fost reparată, dar reapare în versiuni ulterioare
  
- **Testarea regresivă**: repetarea testelor anterioare pentru a ne asigura că eșecurile cunoscute ale versiunilor precedente nu apar în versiuni mai noi

Computer Science



# Acoperirea testelor

- **Testarea tip cutie neagră**: testează funcționalitatea fără a ține seama de structura internă a implementării
- **Testarea tip cutie albă**: ia în considerare structura internă la proiectarea testelor
- **Acoperirea testelor**: măsoară câte părți dintr-un program au fost testate
  - Trebuie să ne asigurăm că fiecare parte a programului a fost testată măcar o dată de un caz de test
  - D.e., ne asigurăm că am executat fiecare ramură în cel puțin un caz de test

Computer Science



# Acoperirea testelor

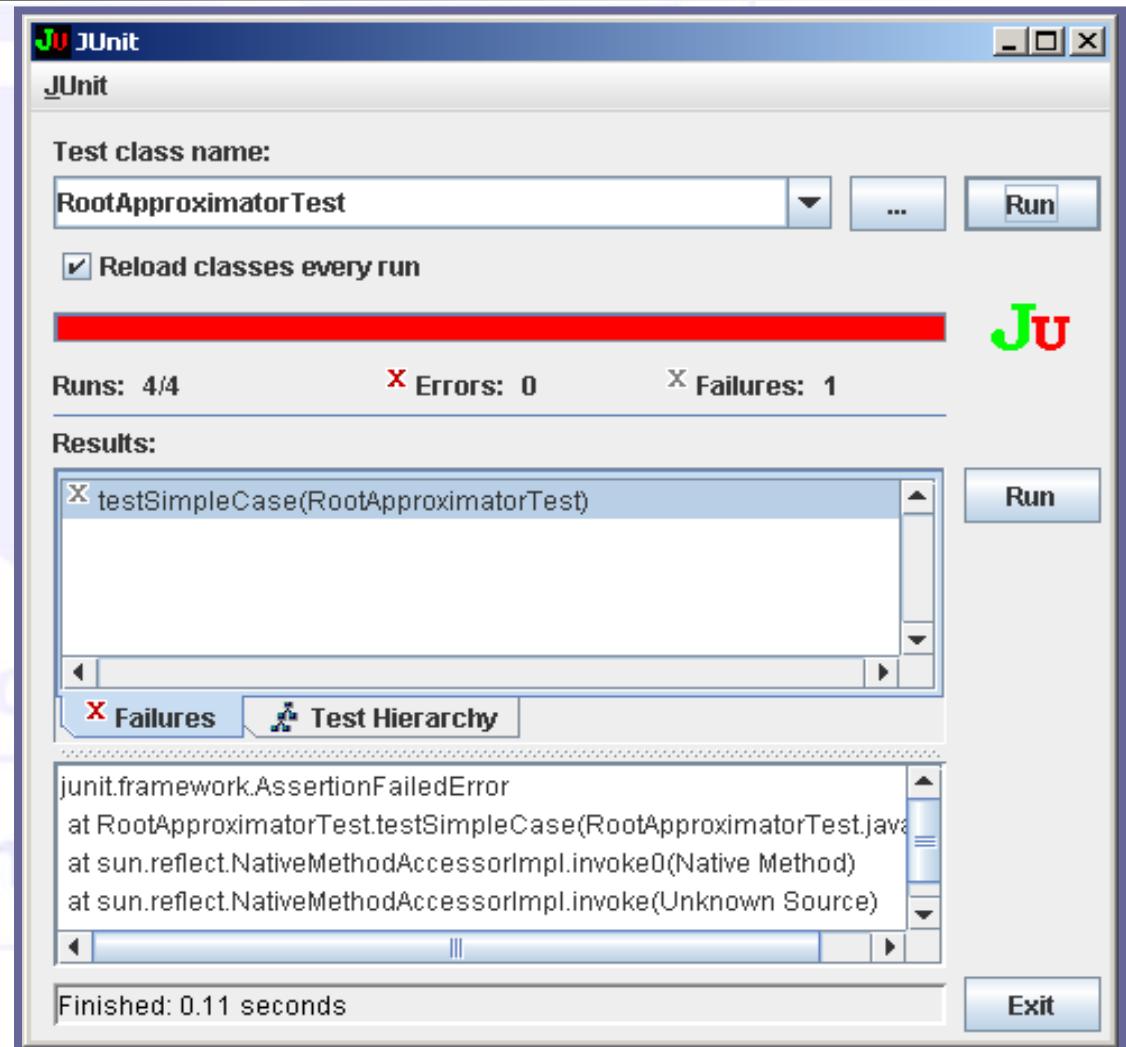
- Sugestie: scrieți primele cazuri de test înainte de a termina scrierea completă a programului → vă permite să intuiți mai bine ce ar trebui să facă programul
- Programele de azi pot fi dificil de testat
  - GUI (folosirea mouse)
  - Conexiunile în rețea (întârzierea și căderile)
  - Există unelte pentru a automatiza testarea în aceste scenarii
  - Principiile de bază ale testării regresive și ale acoperirii complete se mențin

Computer Science



# Testarea unităților cu JUnit

- <http://junit.org>
- Preconstruit în unele IDE cum sunt BlueJ și Eclipse
- Filozofia: ori de câte ori implementăm o clasă, implementăm și o clasă însotitoare, de test
- În dreapta se află un exemplu cu UI Swing UI de lucru cu junit 3.8.1





# Exemplu simplu cu JUnit

```
public class Calculate {  
    public int sum(int var1, int var2) {  
        System.out.println("Adding values: " + var1 + " + " + var2);  
        return var1 + var2;  
    }  
}
```

```
import static org.junit.Assert.assertEquals;  
import org.junit.jupiter.api.Test;  
  
public class CalculateTest {  
    Calculate calculation = new Calculate();  
    int sum = calculation.sum(2, 5);  
    int testSum = 7;  
    @Test  
    public void testSum() {  
        System.out.println("@Test sum(): " + sum + " = " + testSum);  
        assertEquals(sum, testSum);  
    }  
}
```



# Exemplu simplu cu JUnit

eclipse-workspace - JUnitTestingExample/src/CalculateTest.java - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer JUnit ✘

Finished after 0.175 seconds

Runs: 1/1 Errors: 0 Failures: 0

> CalculateTest [Runner: JUnit 5] (0.009 s)

Run As > JUnit Test Alt+Shift+X, T

```
1 CalculateTest
2 Tst
3 MainException
4 TestMultipleCatchBlock
5 AssertDemo
6 Demo
7 MainClass (3)

11     System.out.println("@Test sum(): " + sum + " = " + testSum);
12     assertEquals(sum, testSum);
13 }
14
15 }
16
```

Problems @ Javadoc Declaration Console

<terminated> CalculateTest [JUnit] C:\Program Files\Java\jre1.8.0\_144\bin\javaw.exe (Dec 10, 2018, 12:32:18 PM)

Adding values: 2 + 5  
@Test sum(): 7 = 7



# Trasarea execuției programului

- Mesaje care arată calea urmată de execuție

```
if (status == SINGLE) {  
    System.out.println("status is SINGLE");  
    . . .  
}  
. . .
```

- Neajuns: trebuie eliminate atunci când s-a terminat testarea și repuse înapoi când apare o altă eroare
- Soluția: folosim clasa **Logger** (pentru jurnalizare) pentru a stopa scrierea mesajelor din trasare fără a le elibera din program (**java.util.logging**)



# Jurnalizarea

- Mesajele de jurnalizare pot fi dezactivate la terminarea testării
- Folosim obiectul global `Logger.global`
- Jurnalizăm un mesaj
- Implicit, mesajele jurnalizate se tipăresc. Le inhibăm cu  
`Logger.global.setLevel(Level.OFF);`
- Jurnalizarea poate fi o problemă de gândit (nu trebuie să jurnalizăm nici prea multă informație, nici prea puțină)  
`Logger.global.info("status is SINGLE");`
- Unii programatori preferă depanarea (*debugging*) în locul jurnalizării (*logging*)



# Jurnalizarea

- La trasarea cursului execuției, cele mai importante evenimente sunt intrarea în și ieșirea dintr-o metodă
- La începutul metodei, tipărim parametrii:

```
public TaxReturn(double anIncome, int aStatus) {  
    Logger.global.info("Parameters: anIncome = " + anIncome  
        + " aStatus = " + aStatus);  
    . . .  
}
```

- La sfârșitul metodei, tipărim valoarea returnată:

```
public double getTax() {  
    . . .  
    Logger.global.info("Return value = " + tax);  
    return tax;  
}
```



# Jurnalizarea

- Biblioteca de jurnalizare are un set de nivele predefinite:

|         |                                                                                                         |
|---------|---------------------------------------------------------------------------------------------------------|
| SEVERE  | <i>Cea mai mare valoare</i> ; menită pentru mesaje extrem de importante (d.e. erori de program fatale). |
| WARNING | Destinată mesajelor de avertizare.                                                                      |
| INFO    | Pentru mesaje de execuție informative.                                                                  |
| CONFIG  | Mesaje informative despre setările de configurare/setup.                                                |
| FINE    | Folosit pentru detalii mai fine la depanarea/diagnosticarea problemelor.                                |
| FINER   | Mai în detaliu.                                                                                         |
| FINEST  | <i>Cea mai mică valoare</i> ; cel mai mare grad de detaliu.                                             |

- Pe lângă aceste nivele:
  - Nivelul ALL care activează jurnalizarea tuturor înregistrărilor
  - Nivelul OFF care poate fi folosit la dezactivarea jurnalizării
  - Se pot defini nivele individualizate (vezi documentația Java!)



# Exemplu pentru Logger

```
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
public class LoggerExample {
    private static final Logger LOGGER =
        Logger.getLogger(LoggerExample.class.getName());
    public static void main(String[] args) throws SecurityException,
  IOException {
        LOGGER.info("Logger Name: "+LOGGER.getName());
        LOGGER.warning("Can cause ArrayIndexOutOfBoundsException");
        //An array of size 3
        int []a = {1,2,3};
        int index = 4;
        LOGGER.config("index is set to "+index);
        try{
            System.out.println(a[index]);
        }catch(ArrayIndexOutOfBoundsException ex){
            LOGGER.log(Level.SEVERE, "Exception occur", ex);
        }
    }
}
```



# Avantajele jurnalizării

- Jurnalizarea poate genera informații detaliate despre funcționarea unei aplicații
- După ce a fost adăugată la aplicație, nu mai are nevoie de intervenția umană
- Jurnalele de aplicație pot fi salvate și studiate ulterior
- Prin surprinderea erorilor care nu pot fi raportate utilizatorilor, jurnalizarea poate ajuta în determinarea cauzelor problemelor apărute
- Prin surprinderea mesajelor foarte detaliate și a celor specificate de programatori, jurnalizarea poate ajuta la depanare
- Poate fi o unealtă de depanare acolo unde nu sunt disponibile depanatoarele – adesea aceasta este situația la aplicații distribuite sau multi-fir (*multithreaded*)
- Jurnalizarea rămâne împreună cu aplicația și poate fi folosită oricând se rulează aplicația



# Costurile jurnalizării

- Jurnalizarea adaugă o încărcare suplimentară la execuție datorată generării mesajelor și I/E pe dispozitivele de jurnalizare
- Jurnalizarea adaugă o încărcare suplimentară la programare, pentru că trebuie scris cod suplimentar pentru a genera mesajele
- Jurnalizarea crește dimensiunea codului
- Dacă jurnalele sunt prea "vorbărețe" sau prost formatare, extragerea informației din acestea poate fi dificilă
- Instrucțiunile de jurnalizare pot scădea lizibilitatea codului
- Dacă mesajele de jurnalizare nu sunt întreținute odată cu codul din jur, atunci pot cauza confuzii și deveni o problemă de întreținere
- Dacă nu sunt adăugate în timpul dezvoltării inițiale, adăugarea ulterioară poate necesita un volum mare de muncă pentru modificarea codului



# Depanarea

- Depanator (*debugger*)= program folosit la rularea altui program care permite analizarea comportamentului la execuție al programului rulat
- Depanatorul permite oprirea și repornirea programului, precum și execuția sa pas-cu-pas
- Cu cât sunt mai mari programele, cu atât sunt mai greu de depanat prin simpla jurnalizare
- Depanatoarele pot fi parte a IDE (Eclipse, BlueJ, Netbeans) sau programe separate (JSwat)
- Trei concepte cheie:
  - Puncte de întrerupere (*breakpoints*)
  - Execuție pas-cu-pas (*single-stepping*)
  - Inspectarea variabilelor



# Despre depanatoare

- Programele se întâmplă să aibă erori de logică
- Uneori problema poate fi descoperită imediat
- Alteori trebuie determinată
- Un depanator poate fi de mare ajutor
  - Câteodată este exact unealta necesară
  - Alteori, nu
- Depanatoarele sunt în esență asemănătoare
  - “Dacă știi unul, le știi pe toate”
- Depanatorul permite execuția linie cu linie, instrucțiune cu instrucțiune
- La fiecare pas se pot examina valorile variabilelor



# Despre depanatoare

- Se pot seta puncte de întrerupere (breakpoints) și se poate spune depanatorului să “continue” (să ruleze mai departe la viteza maximă) până când întâlnește următorul punct de întrerupere
  - La următorul punct de întrerupere se poate relua execuția pas cu pas
- Punctele de întrerupere rămân active până când sunt înălăturate
- Execuția este suspendată ori de câte ori se întâlnește un punct de întrerupere
- În depanator, programul rulează la viteza maximă până ajunge la un punct de întrerupere
- La oprirea execuției putem:
  - Inspecta variabile
  - Executa programul linie cu linie, sau continua rularea la viteza maximă până la următorul punct de întrerupere



# Introducere În I/E Java

- Sistemul de I/E este foarte complex
  - Încearcă să facă multe lucruri folosind componente reutilizabile
  - Există de fapt trei sisteme de I/E
    - Cel original din JDK 1.0
    - Unul mai nou începând cu JDK 1.2 care se suprapune și îl înlocuiește parțial pe primul
    - Pachetul `java.nio` din JDK 1.4 este și mai nou
- Efectuarea de operații de I/E cere programatorului să folosească o serie de clase complexe
  - De obicei se creează clase auxiliare cum sunt `StdIn`, `FileIn` și `FileOut` pentru a ascunde această complexitate



# Introducere În I/E Java

## ■ Motivele complexității Java I/E

- Sunt multe tipuri diferite de surse și absorbante (*sinks*)
- Două tipuri diferite de acces la fișiere
  - Acces secvențial
  - Acces aleator
- Două tipuri diferite de formate de stocare
  - Formatat
  - Neformatat
- Trei sisteme de I/E diferite (vechi și noi)
- O mulțime de clase “filtru” sau “modificator”

Computer Science



# Accesul aleatoriu vs. secvențial

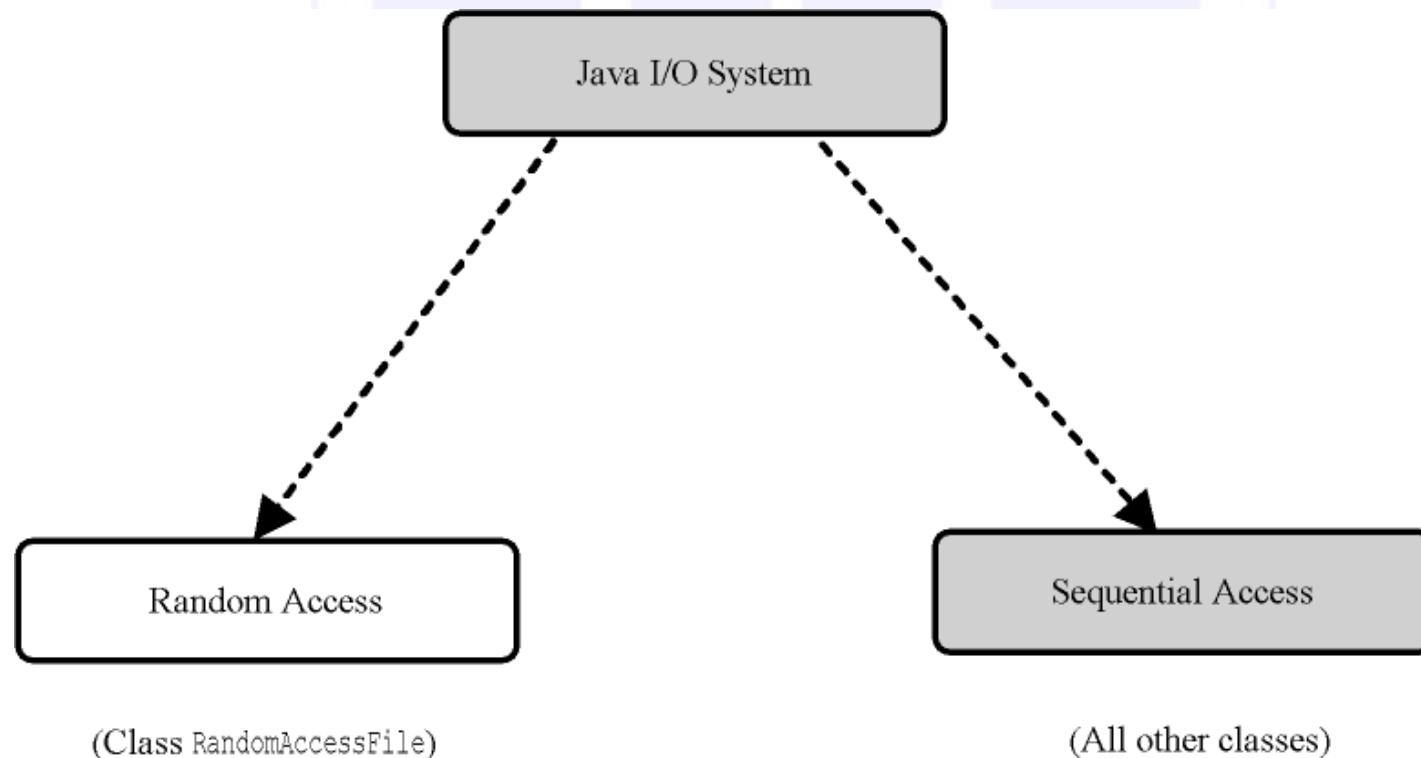
- Accesul secvențial
  - Fișierul este prelucrat octet după octet
  - Poate fi ineficient
- Accesul aleator
  - Permite accesul la locații arbitrate în fișier
  - Doar fișierele disc suportă accesul aleator
    - `System.in` și `System.out` nu-l suportă
  - Fiecare fișier disc are o poziție specială pentru indicatorul de fișier
    - Se poate citi sau scrie la poziția curentă a indicatorului

Computer Science



# Structura sistemului de I/E Java (java.io)

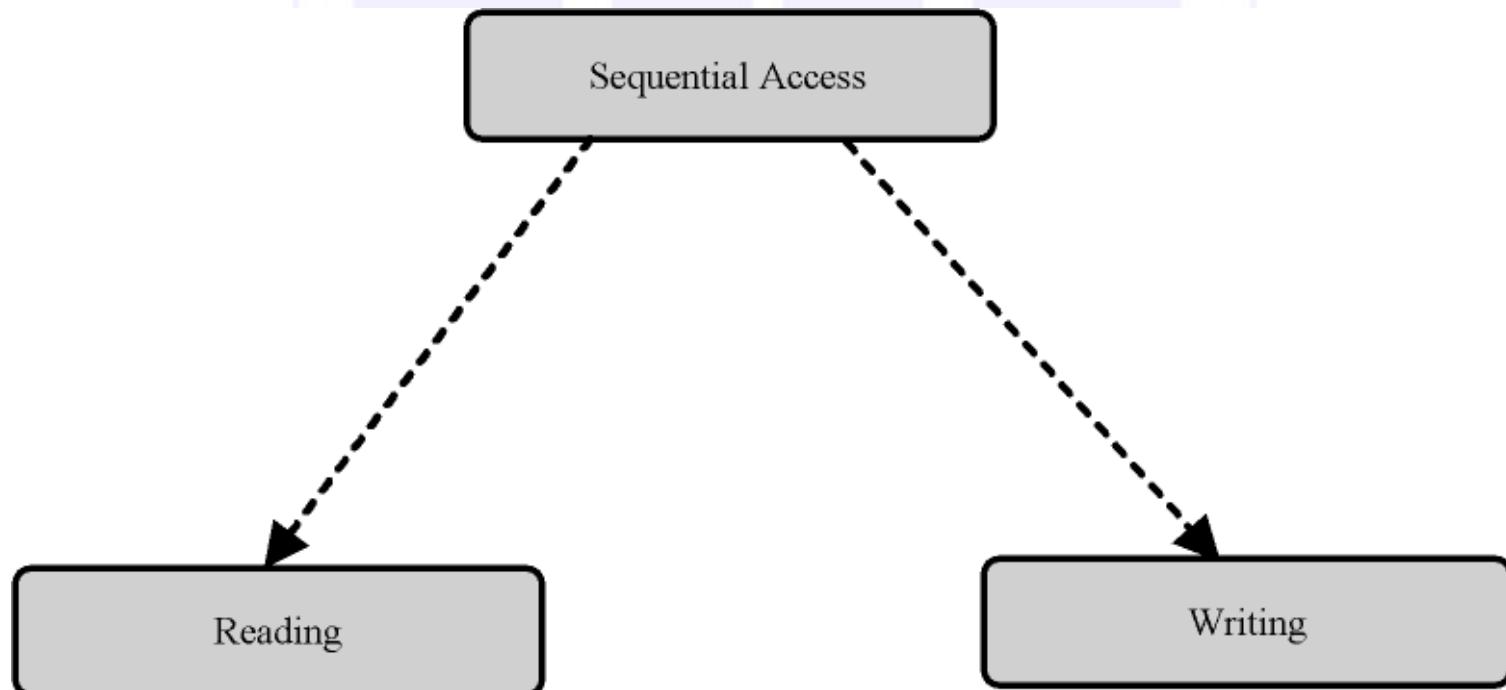
- Sistemul de I/E Java este divizat în clase pentru accesul secvențial și clase pentru accesul aleatoriu (numit și acces direct):





# Structura sistemului de I/E Java (java.io)

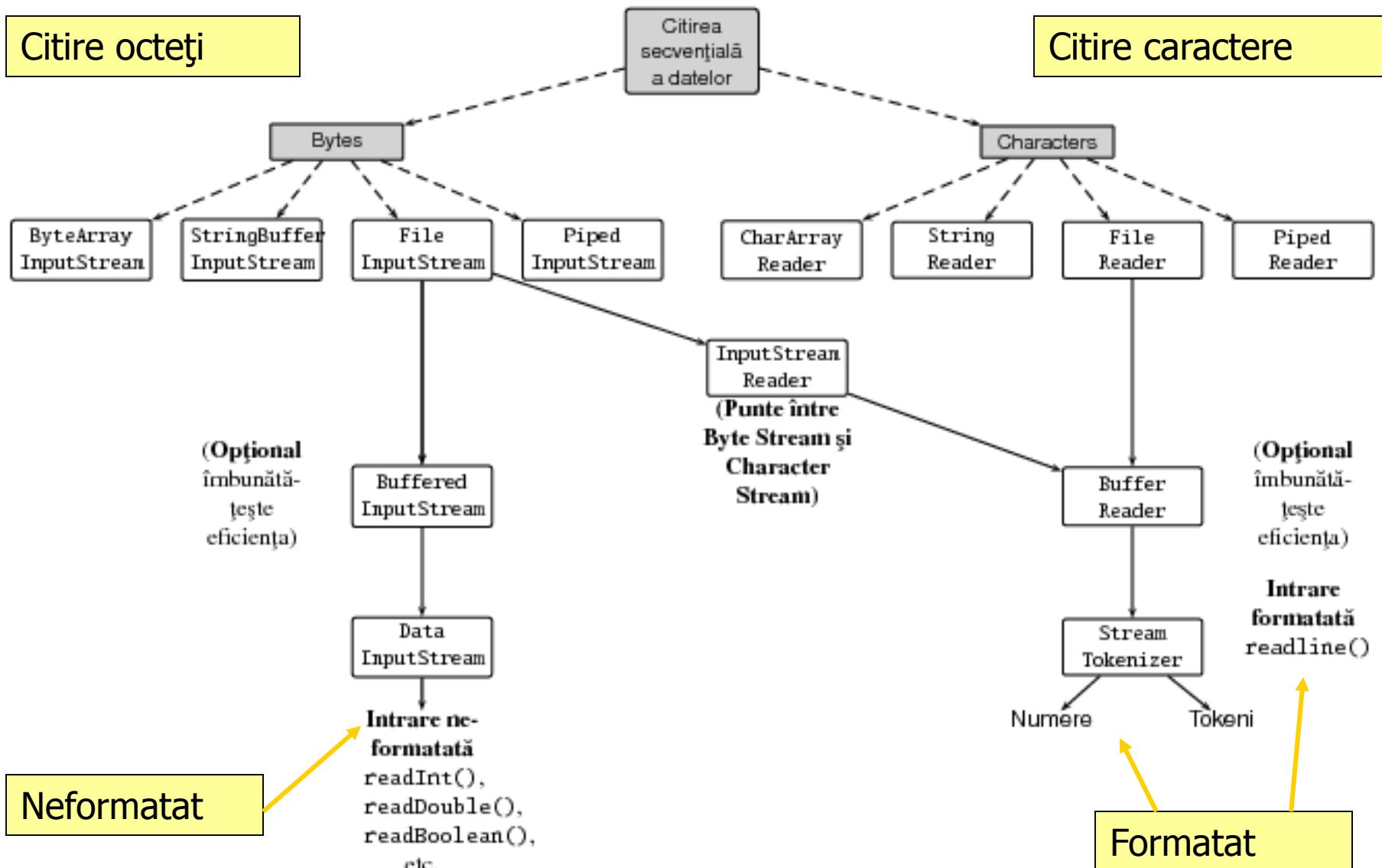
- Accesul secvențial este subîmpărțit în clase pentru citire și clase pentru scriere:





# Clase pentru citirea secvențială a datelor (din java.io)

Citire octeți



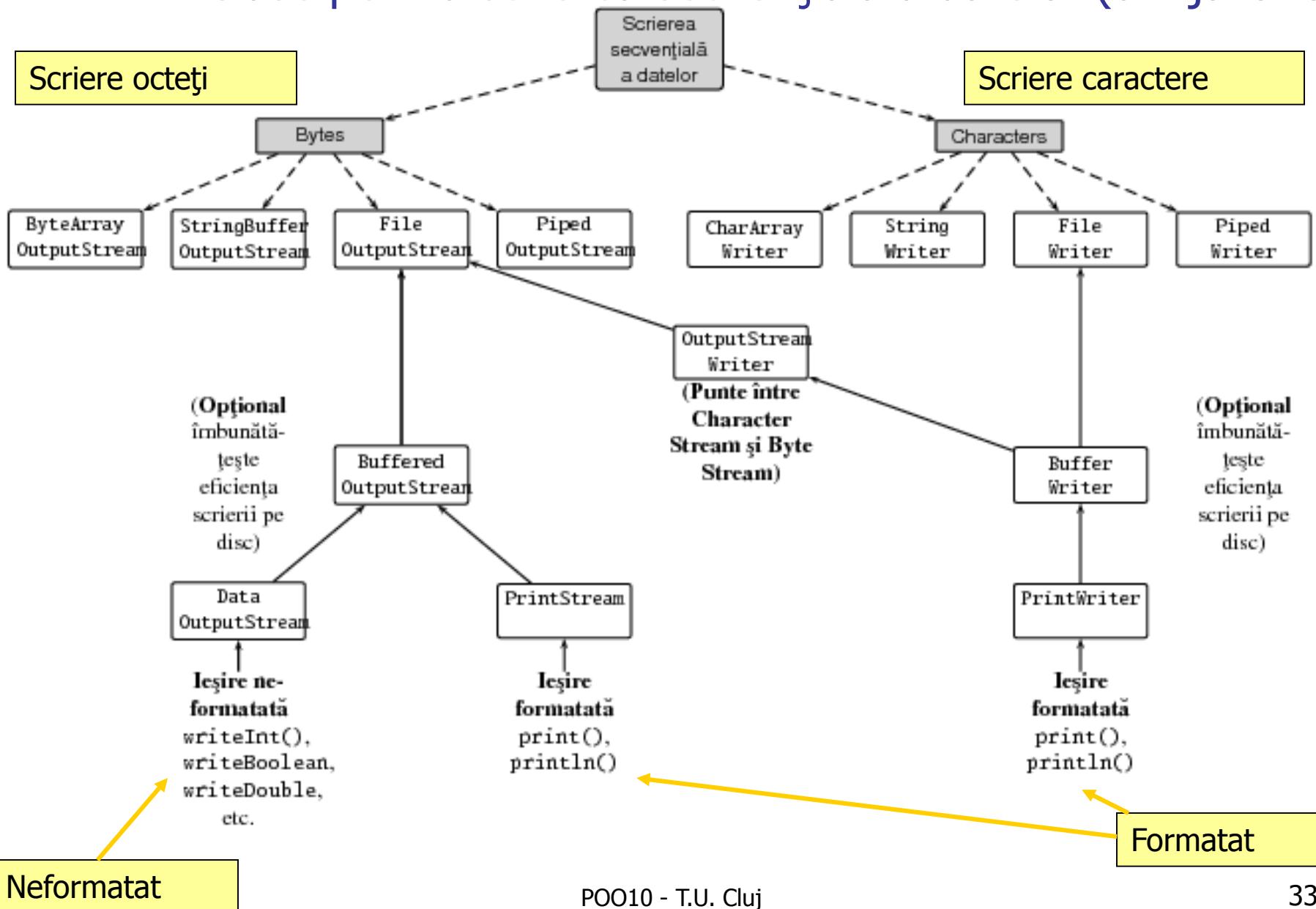
Nefomatat

Formatat



# Clase pentru scrierea secvențială a datelor (din java.io)

Scriere octeți





# Excepții

- Toate clasele de I/E Java aruncă excepții, cum este **FileNotFoundException** și excepția mai generală **IOException**
- Programele Java trebuie să intercepteze explicit excepțiile de I/E în structuri **try / catch** pentru a gestiona problemele de I/E
  - Această structură *trebuie* să trateze **IOException**, care este clasa generală de excepții de I/E
  - Poate trata excepțiile de nivel mai jos separat – cum este cazul cu **FileNotFoundException** – permite programului să ofere utilizatorului informații inteligente și opțiuni în cazul în care nu se găsește un fișier



# Folosirea I/E Java

- Procedura generală pentru folosirea I/E Java este:
  - Creăm o structură **try/catch** pentru excepțiile de I/E
  - Alegem o clasă de intrare sau ieșire pe baza tipului de I/E (formatat sau neformatat, secvențial sau direct) și tipul de flux (stream) de intrare sau ieșire (fișier, conductă [*pipe*], etc.)
  - Împachetăm clasa de intrare sau ieșire într-o clasă tampon (buffer) pentru creșterea eficienței
  - Folosim clase filtru sau modificatoare pentru a translata datele în forma corespunzătoare pentru intrare sau ieșire (d.e., **DataInputStream** sau **DataOutputStream**)



# Exemplu: Citirea de **String**-uri dintr-un fișier secvențial formatat

- Alegem clasa **FileReader** pentru a citi date secvențiale formatare
  - Deschidem fișierul prin crearea unui obiect **FileReader**
  - Împachetăm **FileReader** într-un **BufferedReader** pentru eficiență
  - Citim fișierul cu metoda **BufferedReader** numită **readLine()**
  - Închidem fișierul folosind metoda **close()** a lui **FileReader**
  - Tratăm exceptiile de I/E folosind o structură **try/catch**



# Exemplu

Includem I/E într-o structură  
**try/catch**

Deschidem fișierul prin crearea unui **FileReader** împachetat într-un **BufferedReader**

Citim linii cu **readLine()**

Închidem fișierul cu **close()**

Tratăm exceptiile

```
// Interceptam exceptiile daca apar
try
{
    // Creeaza BufferedReader
    BufferedReader in =
        new BufferedReader( new FileReader(args[0]) );
    // Read file and display data
    while( (s = in.readLine()) != null)
    {
        System.out.println(s);
    }
    // Inchide fisierul file
    in.close();
}
// Intercepteaza FileNotFoundException
catch (FileNotFoundException e)
{
    System.out.println("File not found: " + args[0]);
}
// Interceptează alte IOExceptions
```



# Scanner

TECHNICAL UNIVERSITY

- În loc să citim direct din `System.in` sau dintr-un fișier text folosim un **Scanner**
  - Întotdeauna trebuie să spunem lui **Scanner** ce să citească
  - D.e. îl instanțiem cu o referință pentru a citi din `System.in`

```
java.util.Scanner scanner =  
        new java.util.Scanner(System.in);
```
- Ce anume face **Scanner**?
  - Divizează intrarea în unități gestionabile numite *token-i*

```
Scanner scanner = new Scanner(System.in);  
String userInput = scanner.nextLine();
```

`nextLine()` ia tot ce s-a tastat la consolă până când utilizatorul apasă tasta "Enter"
  - *Token-ii* au mărimea liniilor de intrare și sunt de tipul **String**



# Alte metode din clasa Scanner

| Pentru a citi un:                                                                      | Folosim metoda Scanner             |
|----------------------------------------------------------------------------------------|------------------------------------|
| <code>boolean</code>                                                                   | <code>boolean nextBoolean()</code> |
| <code>double</code>                                                                    | <code>double nextDouble()</code>   |
| <code>float</code>                                                                     | <code>float nextFloat()</code>     |
| <code>int</code>                                                                       | <code>int nextInt()</code>         |
| <code>long</code>                                                                      | <code>long nextLong()</code>       |
| <code>short</code>                                                                     | <code>short nextShort()</code>     |
| <code>String</code> (care apare pe linia următoare, până la '\n')                      | <code>String nextLine()</code>     |
| <code>String</code> (care apare pe linia următoare, până la următorul ' ', '\t', '\n') | <code>String next()</code>         |



# Exceptii pentru Scanner

## ■ **InputMismatchException**

- Aruncata de toate metodele `nextType()`
- Semnificație: token-ul nu poate fi convertit într-o valoare de tipul specificat
- **Scanner** nu avansează la token-ul următor, astfel că acest token poate fi încă regăsit

## ■ Tratarea acestei exceptii

- Preveniți-o
  - Testați token-ul următor folosind o metodă `hasNextType()`
  - Metoda nu avansează, doar verifică tipul token-ului următor
    - `boolean hasNextBoolean()`
    - `boolean hasNextDouble()`
    - `boolean hasNextFloat()`
    - `boolean hasNextInt()`
    - `boolean hasNextLong()`
    - `boolean hasNextShort()`
    - `boolean hasNextLine()`
- Interceptați-o
  - Tratați exceptia o dată interceptată



# Fluxuri (*streams*) de obiecte

- Clasa **ObjectOutputStream** poate salva obiecte pe disc
- Clasa **ObjectInputStream** poate citi obiectele de pe disc înapoi în memorie
- Obiectele sunt salvate în format binar; de aceea folosim fluxuri (streams)
- Fluxul pentru ieșire de obiecte salvează toate variabilele instanță
  - Exemplu: Scrierea unui obiect **BankAccount** într-un fișier

```
BankAccount b = .OF CLUJ-NAPOCA.  
ObjectOutputStream out = new ObjectOutputStream(  
    new FileOutputStream("bank.dat"));  
out.writeObject(b);
```



# Exemplu: citirea unui obiect **BankAccount** dintr-un fișier

- **readObject** returnează o referință la un **Object**
  - Este nevoie să ne reamintim tipurile obiectelor care au fost salvate și să folosim o forțare (*cast*) de tip

```
ObjectInputStream in = new ObjectInputStream(
    new FileInputStream("bank.dat"));
BankAccount b = (BankAccount) in.readObject();
```
- Metoda **readObject** poate arunca o excepție de tipul **ClassNotFoundException**
  - Este o excepție verificată
  - Trebuie fie interceptată, fie declarată



# Scrierea și citirea unui **ArrayList** într-un/dintr-un fișier

## ■ Scrierea

```
ArrayList<BankAccount> a = new ArrayList<BankAccount>();  
// Se adauga mai multe obiecte BankAccount in a  
out.writeObject(a);
```

## ■ Citirea

```
ArrayList<BankAccount> a =  
(ArrayList<BankAccount>) in.readObject();
```



# Serializabil

- Obiectele care sunt scrise într-un flux de obiecte trebuie să aparțină unei clase care implementează interfața **Serializable**

```
class BankAccount implements Serializable {  
    . . .  
}
```

- Interfața **Serializable** nu are metode
- **Serializare**: procesul de salvare a obiectelor într-un flux
  - Fiecărui obiect îi este atribuit un număr de serie pe flux
  - Dacă același obiect este salvat de două ori, a doua oară se salvează numai numărul de serie
  - La citire, numerele de serie duplicate sunt restaurate ca referințe la același obiect



# Exemplu Serializare/Deserializare

```
import java.io.*;
public class Angajat implements Serializable {
    transient int a; //a nu va fi serializat datorita lui transient
    static int b; //b nu va fi serializat deoarece este static
    String name;
    int age;

    public Angajat(String name, int age, int a, int b)
    {
        this.name = name;
        this.age = age;
        this.a = a;
        this.b = b;
    }
}
```



# Exemplu Serializare/Deserializare

```
import java.io.*;
public class ExempluSerializare {
    public static void afisareDate(Angajat object1) {
        System.out.println("name = " + object1.name);
        System.out.println("age = " + object1.age);
        System.out.println("a = " + object1.a);
        System.out.println("b = " + object1.b);
    }
    public static void main(String[] args) {
        Angajat object = new Angajat("Pop Dorel", 20, 2, 1000);
        String filename = "angajat.dat";
        // Serializare
        try {
            // Salveaza obiectul in fisier
            FileOutputStream file = new FileOutputStream (filename);
            ObjectOutputStream out = new ObjectOutputStream (file);
            out.writeObject(object);
            out.close(); file.close();
            System.out.println("Obiect serializat\n" + "Date inainte de deserializare:");
            afisareDate(object);
            object.b = 2000; // se schimba valoarea variabilei statice
        }catch (IOException ex) {
            System.out.println("IOException is caught");
        }
    }
}
```



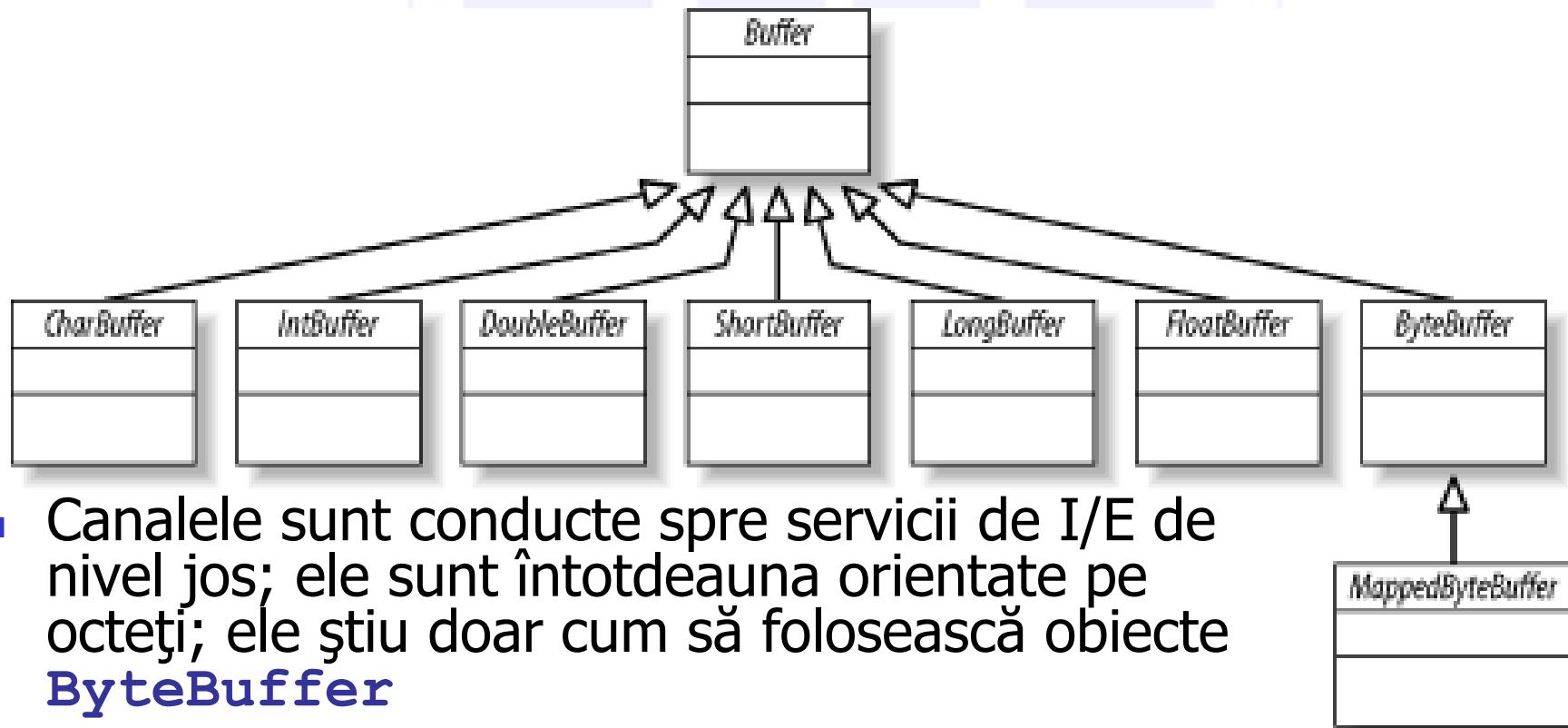
# Exemplu Serializare/Deserializare

```
// Deserializare
object = null;
try {
    // Citeste obiect din fisier
    FileInputStream file = new FileInputStream (filename);
    ObjectInputStream in = new ObjectInputStream (file);
    // Deserializeaza obiect
    object = (Angajat)in.readObject();
    in.close();
    file.close();
    System.out.println("Obiect deserializat\n Date dupa deserializare.");
    afisareDate(object);
}
catch (IOException ex) {
    System.out.println("IOException is caught");
}
catch (ClassNotFoundException ex) {
    System.out.println("ClassNotFoundException is caught");
}
}
```



# Zone tampon și canale

- Zonele tampon au fost create în primul rând pe post de containere pentru datele (de tipuri primitive) trimise/recepționate pe/de pe canale



- Canalele sunt conducte spre servicii de I/E de nivel jos; ele sunt întotdeauna orientate pe octeți; ele știu doar cum să folosească obiecte **ByteBuffer**



# Vederi pentru Buffer

- Presupunem că avem un fișier care conține caractere Unicode stocate ca valori pe 16 biți (codificare UTF-16 nu UTF-8)
  - UTF = Unicode Transformation Format
- Pentru a citi o bucată din acest fișier în zona tampon putem crea o vedere **CharBuffer** a octetilor respectivi:

```
CharBuffer charBuffer = byteBuffer.asCharBuffer();
```

  - Creează o vedere a **ByteBuffer** care se comportă ca un **CharBuffer** (combină fiecare pereche de octeți din tampon într-o valoare caracter pe 16 biți)
- Clasa **ByteBuffer** are metode de acces ad-hoc la valorile primitive
  - D.e., pentru a accesa ca întreg patru octeți dintr-o zonă tampon

```
int fileSize = byteBuffer.getInt();
```



# Exemplu de vederi pentru Buffer

```
import java.nio.*;
public class Buffers {
    public static void main(String[] args)  {
        try {
            float[] floats = {
                6.612297E-39F, 9.918385E-39F,
                1.1093785E-38F, 1.092858E-38F,
                1.0469398E-38F, 9.183596E-39F
            }
            ByteBuffer bb =
                ByteBuffer.allocate(floats.length * 4 );
```

```
        FloatBuffer fb = bb.asFloatBuffer();
        fb.put(floats);
        CharBuffer cb = bb.asCharBuffer();
        System.out.println(cb.toString());
    } catch (Exception e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
}
```

|             |                                                  |              |               |              |               |              |
|-------------|--------------------------------------------------|--------------|---------------|--------------|---------------|--------------|
| FloatBuffer | 6.612297E-39                                     | 9.918385E-39 | 1.0193785E-38 | 1.092858E-38 | 1.0469398E-38 | 9.183596E-39 |
| CharBuffer  | H                                                | e            | l             | l            | o             | w            |
| ByteBuffer  | 00480065006C006C006F00200077006F0072006C00640021 |              |               |              |               | !            |



# Interschimbarea octetilor

- *Endian-ness* : ordinea de combinare a octetilor pentru a forma valori numerice mai mari
  - Când octetul cel mai semnificativ ca ordine numerică este primul stocat în memorie (la adresa mai mică) avem ordinea *big-endian*
  - Cazul opus, în care cel mai puțin semnificativ octet apare primul, este *little-endian*



little endian

big endian



# Vederi ale Buffer și Endian-ness

- Fiecare obiect tampon are o setare a *ordinii octeților*
  - Cu excepția lui **ByteBuffer**, proprietatea poate fi numai citită și nu se poate schimba
- Setarea ordinii octeților la obiectele **ByteBuffer** poate fi modificată oricând
  - Aceasta afectează ordinea rezultată pentru orice vederi create pentru acel obiect **ByteBuffer**
  - Dacă datele Unicode din fișier au fost codificate ca UTF-16LE (little-endian) trebuie să setăm ordinea pentru **ByteBuffer** înainte de a crea vederea **CharBuffer**:

```
byteBuffer.order(ByteOrder.LITTLE_ENDIAN);
CharBuffer charBuffer = byteBuffer.asCharBuffer();
```
- Noua vedere moștenește ordinea lui **ByteBuffer**
- Setarea ordinii octeților la momentul apelului *affectează* modul de combinare pentru formarea valorii returnate sau divizate pentru ceea ce este stocat în zona tampon



# Citiri distribuitoare (*scatering*) și scrieri colectoare (*gathering*)

- D.e., cu o singură cerere de citire de pe canal putem pune primii 32 octeți în tamponul **header**, următorii 768 octeți în tamponul **colorMap** și restul în **imageBody**
- Canalul umple fiecare tampon pe rând până când toate sunt pline sau nu mai sunt date de citit

```
    . . .
ByteBuffer header = ByteBuffer.allocate(32);
ByteBuffer colorMap = ByteBuffer(256 * 3);
ByteBuffer imageBody = ByteBuffer(640 * 480);
ByteBuffer[] scatterBuffers = { header, colorMap,
    imageBody };
fileChannel.read(scatterBuffers);
```



# Transferuri directe pe canale

- Transferul pe canal permite interconectarea a două canale astfel încât datele să fie transferate direct dintr-un canal în celălalt fără intervenții suplimentare
- Deoarece metodele `transferTo()` și `transferFrom()` aparțin clasei `FileChannel`, trebuie ca sursa și destinația unui transfer pe canal să fie obiecte `FileChannel` (d.e., nu se poate transfera de la un *socket* la altul)
- Celălalt capăt poate fi orice `ReadableByteChannel` sau `WritableByteChannel`, după nevoi



# Exemplu de transfer direct între canale

```
import java.nio.*;
import java.nio.channels.*;
import java.io.*;
public class DirectChannelTransfer {
    public static void main(String args[]) throws IOException {
        // verifica argumentele de pe linia de comanda
        if (args.length != 2) {
            System.err.println("Lipsesc numele de fisiere");
            System.exit(1);
        }
        // get channels
        FileInputStream fis = new FileInputStream(args[0]);
        FileOutputStream fos = new FileOutputStream(args[1]);
        FileChannel fcin = fis.getChannel();
        FileChannel fcout = fos.getChannel();
        // executa copierea fisierului
        fcin.transferTo(0, fcin.size(), fcout);
        // incheie
        fcin.close();
        fcout.close();
        fis.close();
        fos.close();
    }
}
```

Metoda **transferTo** transferă octeți din canalul sursă (**fcin**) în canalul destinație specificat (**fcout**). Transferul este executat tipic *fără citiri și scrieri explicate la nivel utilizator* pe canal.

Computer Science



# Expresii regulate

- Expresiile regulate (`java.util.regex`) sunt parte a NIO
- Clasa `String` știe de expresii regulate prin adăugarea următoarelor metode:

```
package java.lang;
public final class String implements java.io.Serializable,
    Comparable, CharSequence
{
    // Lista parțială din API
    public boolean matches (String regex)
    public String [] split (String regex)
    public String [] split (String regex, int limit)
    public String replaceFirst (String regex, String
                               replacement)
    public String replaceAll (String regex, String
                               replacement)
}
```



# Exemple de expresii regulate

```
public static final String VALID_EMAIL_PATTERN =
    "^[_A-Za-z0-9-\\\\+]+(\\.[_A-Za-z0-9-]+)*@[\\A-Za-z0-9-]+(\\.[A-Za-z0-9]+)*(\\.[A-Za-z]{2,})$";
...
if (emailAddress.matches (VALID_EMAIL_PATTERN))
{
    addEmailAddress (emailAddress);
}
else
{
    throw new IllegalArgumentException (emailAddress);
}

// imparte sirul lineBuffer (care contine o serie de valori separate prin
// virgule) in subsiruri si returneaza sirurile respective intr-un tablou
String [] tokens = lineBuffer.split ("\\s*,\\s*");
```



# Exemple de expresii regulate

```
public static final String VALID_EMAIL_PATTERN =  
    "^[_A-Za-z0-9-\\\\+]+(\\.[_A-Za-z0-9-]+)*@[ ]+  
    \"[A-Za-z0-9-]+(\\.[A-Za-z0-9-]+)*(\\.\\.[A-Za-z]{2,})$";
```

|                     |                                                                                                 |
|---------------------|-------------------------------------------------------------------------------------------------|
| ^                   | începutul liniei                                                                                |
| [_A-Za-z0-9-\\\\+]+ | trebuie sa înceapă cu String-ul din [ ], trebuie să conțină unul sau mai multe (+)              |
| (                   | început grup #1                                                                                 |
| \\.[_A-Za-z0-9-]+   | urmat de un punct "." și String-ul din paranteze [ ], trebuie să conțină unul sau mai multe (+) |
| )*                  | sfârșit grup #1, acest grup este optional (*)                                                   |
| @                   | trebuie să conțină simbolul "@"                                                                 |
| [A-Za-z0-9-]+       | urmat de String-ul din paranteze [ ], trebuie să conțină unul sau mai multe (+)                 |
| (                   | început grup #2                                                                                 |
| \\.\\.[A-Za-z0-9-]+ | urmat de un punct "." și String-ul din paranteze [ ], trebuie să conțină unul sau mai multe (+) |
| )*                  | sfârșit grup #2, acest grup este optional (*)                                                   |
| (                   | început grup #3                                                                                 |
| \\.\\.[A-Za-z]{2,}  | urmat de punct "." și String-ul din paranteze [ ], cu lungimea minimă 2                         |
| )                   | sfârșit grup #3                                                                                 |
| \$                  | sfârșitul liniei                                                                                |



# Exemplu cu expresii regulate

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class EmailValidator {
    private Pattern pattern;
    private Matcher matcher;
    private static final String EMAIL_PATTERN =
        "^[_A-Za-z0-9-\\\\+]+(\\.[_A-Za-z0-9-\\\\+])*@" +
        "[A-Za-z0-9-\\\\+](\\. [A-Za-z0-9]+)* (\\. [A-Za-z]{2,})$";
    public EmailValidator() {
        pattern = Pattern.compile(EMAIL_PATTERN);
    }
    public boolean validate(final String hex) {
        matcher = pattern.matcher(hex);
        return matcher.matches();
    }
}
```



# Programare orientată pe obiecte

## 1. Fire de lucru (*Threads*) în Java



# Utilitatea firelor de lucru

- Acolo unde trebuie să se întâmple mai multe lucruri simultan
  - D.e., o aplicație multimedia poate necesita ca procesele audio, video și de control să se execute în paralel
  - Există adesea perioade de așteptare a răspunsului sistemelor de I/E mai lente, timp în care procesorul poate face altceva
- Programe cum sunt sistemele server/client sunt mult mai ușor de proiectat și scris folosind fire de lucru
- Algoritmi matematici, cum sunt sortarea, căutarea numerelor prime etc. utilizând prelucrarea paralelă
- Pe sistemele multi-procesor, mașinile virtuale Java pot rula firele de lucru pe procesoare diferite și pot obține astfel prelucrare paralelă adevărată și creșteri de performanțe semnificative față de platformele mono-procesor



# Multithreading în Java

- Toate programele Java în afara aplicațiilor simple cu intrare-iesire pe consolă sunt aplicații multithreading
- Procesele grele (**heavyweight**) se rulează direct sub sistemul de operare al mașinii locale și pot conține mai multe subprocese
- Procesele ușoare (**lightweight**) conțin un singur proces (curs secvențial) care este lansat din procesul principal, dar cu **fire de lucru (threaduri)** multiple
- Threadurile sunt fire de lucru paralele care rulează înnăuntrul unui program
  - Ele partajează o zonă de memorie comună
- **Multithreading** în Java se referă la un program care execută simultan mai multe threaduri



# Proprietățile firelor de lucru în Java

- Fiecare fir de lucru își începe execuția la o locație bine cunoscută, predefinită
- Fiecare fir de lucru își execută codul începând de la locația de start, într-o secvență ordonată, predefinită (pentru un set de date de intrare dat)
- Fiecare fir de lucru își execută codul independent de celelalte fire de lucru din program
- Firele de lucru par a avea un anumit grad de simultaneitate în execuție
- Firele de lucru au acces la diferite tipuri de date

Computer Science



# Crearea unui Thread

- Un fir de lucru (*thread*) în Java începe prin crearea unei instanțe a clasei `java.lang.Thread`
- Metodele din clasa `Thread` pentru manipularea firelor de execuție sunt, de exemplu:
  - `start()`
  - `yield()`
  - `sleep()`
  - `run()`
- Acțiunea firului de execuție începe la invocarea metodei `run()`
- La apelul metodei `run()` se crează o nouă stivă de apel pentru threadul executat
  - În Java, fiecare thread are propria stivă de apeluri



# Crearea unui Thread

- Definirea și instanțierea unui thread poate fi făcută în unul din cele două feluri:
  - Extinderea clasei `java.lang.Thread`
  - Implementarea interfeței `Runnable`
- Singurul motiv pentru care are sens să se extindă clasa `Thread` este cazul când se dorește realizarea unei versiuni mai specialize de la clasa `Thread`
  - Atunci când dorește un comportament specializat al firului de execuție
- În restul cazurilor (majoritatea cazurilor) când se dorește doar să se specifice ce anume trebuie să execute threadul, se definește o clasă care implementează interfața `Runnable`



# Definirea unui Thread

- Prin extinderea clasei `java.lang.Thread`
  - Modul cel mai simplu de definire a codului de rulat într-un thread separat este:
    - Extinderea clasei `Thread`
    - Suprascrierea metodei `run()`
  - Exemplu:
- ```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Important job running in MyThread");  
    }  
}
```
- Limitarea acestei abordări este că noua clasă nu va mai putea extinde vreo altă clasă din moment ce a extins clasa `Thread`



# Definirea unui Thread

- Prin implementarea interfeței `java.lang.Runnable`
  - Această variantă oferă flexibilitatea de a extinde orice altă clasă, păstrându-și proprietatea de a putea fi executată într-un fir de lucru separat
  - Exemplu:
- ```
class MyRunnable implements Runnable {  
    public void run() {  
        System.out.println("Important job running in MyRunnable");  
    }  
}
```
- Indiferent de modalitatea aleasă, rezultă cod ce poate fi executat într-un fir de lucru separat



# Instantierea unui Thread

- Fiecare thread începe prin instantierea unui obiect de clasă **Thread**
  - Indiferent cum s-a implementat metoda `run()`, prin extinderea clasei **Thread** sau prin implementarea interfeței **Runnable**, este nevoie de un obiect de tipul **Thread** care să facă treaba
- Pentru varianta când s-a extins clasa **Thread**, instantierea este simplă:  
`MyTread t = new MyThread();`
- Pentru varianta când s-a implementat interfața **Runnable**, este nevoie de următorii pași:  
`MyRunnable r = new MyRunnable();  
Thread t = new Thread(r);`
  - În acest caz, obiectul **Runnable** este dat ca și argument al constructorului **Thread** pentru a ști unde se află implementarea metodei `run()`



# Instantierea unui Thread

- Același obiect **Runnable** poate fi pasat ca argument la mai multe threaduri, de exemplu:

```
public class TestThreads {  
    public static void main (String [] args) {  
        MyRunnable r = new MyRunnable();  
        Thread foo = new Thread(r);  
        Thread bar = new Thread(r);  
        Thread bat = new Thread(r);  
    }  
}
```

- Făcând astfel, mai multe threaduri vor rezolva simultan aceeași problemă
- Până aici avem o instanță thread și știm care metodă **run()** se va executa, dar nu avem încă o execuție
- Pentru execuție e nevoie de invocarea metodei **start()**



# Începerea unui thread

- Pentru începerea unui thread se apelează:

`t.start();`

- În acest moment se creează și o nouă stivă de apeluri asociată threadului `t`
- Doar din momentul apelului metodei `start()` threadul este considerat în viață (*alive*), chiar dacă metoda `run()` este posibil să nu se fi executat încă
- Un thread este considerat mort (*no longer alive*) după ce metoda `run()` și-a terminat execuția
- Pentru a verifica starea unui thread (dacă metoda `run()` și-a terminat sau nu execuția) se poate folosi metoda:

`t.isAlive()`



# Începerea unui thread

- Ce se întâmplă concret la apelul metodei `start()`:
  - Un nou thread începe, având o stivă nouă de apeluri (fiecare thread are stiva proprie de apeluri)
  - Starea thread-ului se schimbă de la `new` la `Runnable`
  - Când thread-ul prinde ocazia să se execute, metoda `run()` se va rula
- Exemplu:

```
class FooRunnable implements Runnable {  
    public void run() {  
        for(int x = 0; x < 3; x++)  
            System.out.println("Runnable running");  
    }  
}
```



# Începerea unui thread

```
public class TestThreads {  
    public static void main (String [] args) {  
        FooRunnable r = new FooRunnable();  
        Thread t = new Thread(r);  
        t.start();  
    }  
}
```

## Rezultate afisate:

Runnable running  
Runnable running  
Runnable running



# Începerea unui thread

- Pentru a ști care thread se execută, putem interoga apelând metoda `getName()` din clasa `Thread`
- În următorul exemplu vom da nume threadului și îl vom afișa în metoda `run()` :

```
class NameRunnable implements Runnable {  
    public void run() {  
        System.out.println("NameRunnable running");  
        System.out.println("Run by " + Thread.currentThread().getName());  
    }  
}  
  
public class NameThread {  
    public static void main (String [] args) {  
        NameRunnable nr = new NameRunnable();  
        Thread t = new Thread(nr);  
        t.setName("Fred");  
        t.start();  
    }  
}
```

## Rezultate afișate:

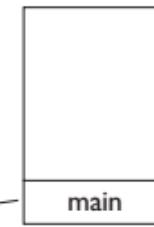
NameRunnable running  
Run by Fred



# Începerea unui thread

## ■ Procesul de începere a unui thread:

```
public static void main(String [] args) {  
    // running  
    // some code  
    // in main()  
    method2();  
    // running  
    // more code  
}  
  
static void method2() {  
    Runnable r = new MyRunnable();  
  
    Thread t = new Thread(r);  
  
    t.start();  
    // do more stuff  
}
```



1) main() begins



2) main() invokes method2()



3) method2() starts a new thread



# Rularea mai multor threaduri

- În continuare este prezentat un exemplu cu o singură interfață Runnable și trei fire de lucru

```
class NameRunnable implements Runnable {  
    public void run() {  
        for (int x = 0; x < 3; x++) {  
            System.out.println("Run by " + Thread.currentThread().getName() + ", x is " + x);  
        }  
    }  
}  
  
public class ManyNames {  
    public static void main(String [] args) {  
        NameRunnable nr = new NameRunnable();  
        Thread one = new Thread(nr);  
        Thread two = new Thread(nr);  
        Thread three = new Thread(nr);  
        one.setName("Fred"); two.setName("Lucy"); three.setName("Ricky");  
        one.start(); two.start(); three.start();  
    }  
}
```

## **Rezultate afisate:**

Run by Lucy, x is 0  
Run by Fred, x is 0  
Run by Ricky, x is 0  
Run by Fred, x is 1  
Run by Fred, x is 2  
Run by Lucy, x is 1  
Run by Ricky, x is 1  
Run by Lucy, x is 2  
Run by Ricky, x is 2



# Rularea mai multor threaduri

- Atenție: aceste rezultate au fost obținute la rularea pe o anumită mașină, **dar această ordine de execuție nu este garantată!**
  - Nu există nici o garanție în specificațiile Java că threadurile își vor începe execuția în ordinea în care a fost apelată metoda `start()`
  - Nu există garanția că o dată ce un thread și-a început execuția, o va continua până la final
  - Și nici garanția că o buclă se va termina de executat înainte ca un alt thread să înceapă
- Singura garanție este următoarea:
  - Fiecare thread va începe execuția și fiecare thread o va finaliza



# Rularea mai multor threaduri

- În interiorul unui thread lucrurile se petrec într-o ordine predictibilă, dar acțiunile threadurilor multiple pot fi amestecate într-o ordine neprevăzută
- Dacă același cod se rulează de mai multe ori, sau pe mașini diferite, rezultatul poate să fie diferit
- De exemplu, dacă modificăm numărul de iteratii la 400:

```
class NameRunnable implements Runnable {  
    public void run() {  
        for (int x = 0; x < 400; x++) {  
            System.out.println("Run by "+  
                Thread.currentThread().getName() + ", x is " + x);  
        }  
    }  
}
```

## Rezultate afisate:

...  
Run by Fred, x is 345  
Run by Ricky, x is 313  
Run by Lucy, x is 341  
Run by Ricky, x is 314  
Run by Lucy, x is 342  
Run by Ricky, x is 315  
Run by Fred, x is 346  
Run by Lucy, x is 343  
Run by Fred, x is 347  
...



# Ordinea execuției threadurilor

- Planificatorul ordinii de execuție a threadurilor (*Thread Scheduler*) ține de JVM
  - Decide care thread să se execute la un anumit moment de timp (din multimea threadurilor eligibile)
  - Scoate threadurile din starea de ***running*** la terminarea execuției lor
- O singură stivă de apeluri poate fi executată la un moment dat pe o unitate de procesare
- Există un comportament de coadă în planificarea ordinii de execuție, în sensul că o dată ce un thread și-a terminat bucata lui de rulat, este pus în capătul cozii unde își așteaptă din nou rândul pentru execuție
- Chiar dacă nu putem controla ordinea de execuție, există totuși unele unelte pentru a influența această ordine



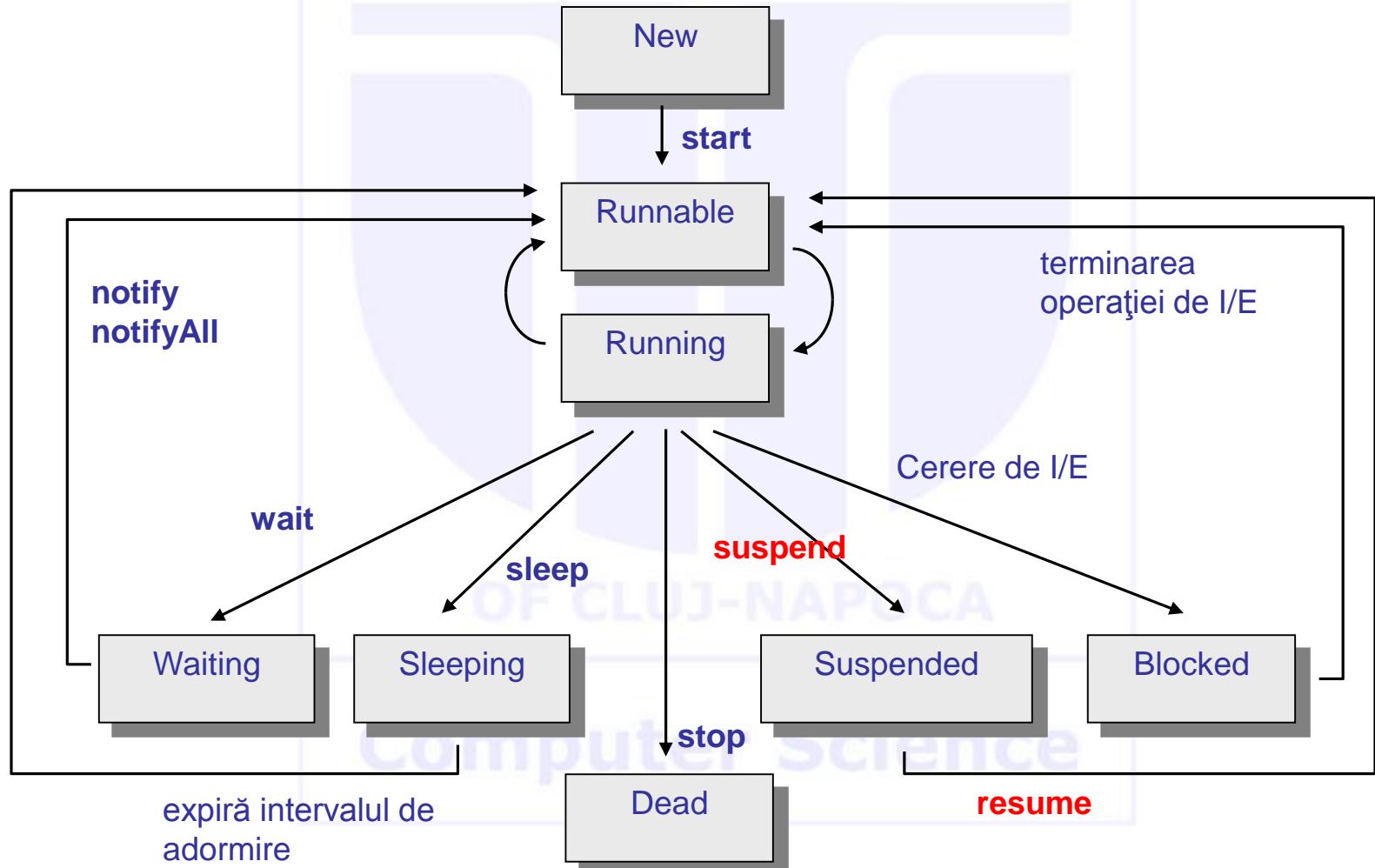
# Ordinea execuției threadurilor

Câteva metode care ajută la influențarea ordinii de execuție a threadurilor sunt:

- Din clasa `java.lang.Thread` :
  - `public static void sleep(long millis) throws InterruptedException`
  - `public static void yield()`
  - `public final void join() throws InterruptedException`
  - `public final void setPriority(int newPriority)`
- Din clasa `java.lang.Object`:
  - `public final void wait() throws InterruptedException`
  - `public final void notify()`
  - `public final void notifyAll()`



# Ciclul de viață al unui fir de lucru





# Starea threadurilor

- Stările pe care un thread le poate avea sunt:
  - **new** – instanța de thread a fost creată, dar încă nu a fost apelată metoda start()
  - **Runnable** – în momentul apelării metodei start(), threadul intră în această stare în care este eligibil pentru execuție
  - **running** – un thread intră în starea de running când planificatorul ordinii de execuție alege acest thread pt execuție; acesta este momentul în care metoda run() a threadul se execută
  - **waiting/blocked/sleeping** – instanța este în viață (alive) dar nu e runnable; el poate reveni la starea de runnable mai târziu, dacă, de exemplu un anumit eveniment este interceptat (I/E, etc.)
  - **dead** – când metoda run() a threadului s-a terminat execuția
- Responsabilitatea planificatorului de threaduri este de a schimba starea threadului



# Thread.sleep()

- **Thread.sleep()** este o metodă statică din clasa **Thread** care pune în pauză un fir de lucru din interiorul căruia se face invocarea metodei
  - Pune în pauză threadul pentru un timp egal cu numărul de milisecunde dat ca argument
  - Remarcați că metoda poate fi invocată dintr-un program obișnuit pentru a insera o pauză în singurul fir al programului respectiv
    - Pentru o aplicație obișnuită (single-threaded), metoda **run()** din thread corespunde metodei **main()**
- Metoda **sleep()** aruncă o excepție de tipul **InterruptedException** atunci când un fir "adormit" este întrerupt
  - Interceptează excepția
  - Termină firul de lucru



# Thread.sleep()

- Prevenirea execuției unui thread folosind metoda **Thread.sleep()**

- Exemplu modificat unde forțăm alternarea execuției threadurilor

```
class NameRunnable implements Runnable {  
    public void run() {  
        for (int x = 0; x < 3; x++) {  
            System.out.println("Run by "  
                + Thread.currentThread().getName());  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException ex) { }  
        }  
    }  
}
```

## Rezultate afisate:

Run by Ricky  
Run by Lucy  
Run by Fred  
Run by Lucy  
Run by Ricky  
Run by Fred  
Run by Ricky  
Run by Fred  
Run by Lucy

- De reținut că ieșirea nu este repetitivă, pentru că nu se știe cu certitudine cât timp durează rularea thread-ului până când este pus în starea de **sleep**



# Terminarea firelor de lucru

- Un fir de lucru se termină la terminarea metodei sale `run()`
- Notificați firul de lucru că ar trebui să își înceteze execuția folosind  
`t.interrupt();`
- `interrupt()` nu face ca firul de lucru să se termine – metoda doar setează un câmp boolean în structura de date a lui
- Java nu forțează terminarea unui fir atunci când acesta este întrerupt
- Este treaba firului de lucru ce anume face atunci când este întrerupt
- Întreruperea reprezintă un mecanism general pentru a "obține atenția" firului de lucru întrerupt



# Terminarea firelor de lucru

- Metoda `run()` ar trebui să verifice ocazional dacă firul de lucru a fost întrerupt
  - Folosiți metoda `interrupted()`
  - Un fir de lucru care a fost întrerupt ar trebui să elibereze resursele pe care le folosește, să "curețe" și să își înceteze execuția

```
public void run()
{
    for(int i=1; i<= REPETITIONS && !Thread.interrupted(); i++)
    {
        //Do work
    }
    //Clean up
}
```



# Prioritățile threadurilor și yield()

- Pentru a înțelege cum funcționează metoda `yield()` trebuie să înțelegem mai întâi conceptul de prioritate a threadurilor
- Threadurile se execută într-o anumită ordine (cu o anumită prioritate)
- Prioritatea este de obicei reprezentată printr-un număr de la 1 (minimă) la 10 (maximă)
- Planificatorul de threaduri se bazează în cazul majorității mașinilor virtuale pe o planificare bazată pe priorități, lucru ce implică un fel de secționare temporală (*time slicing*)
  - Prin folosirea secționării temporale, fiecărui thread i se alocă un anumit timp pentru execuție, după care este trimis în starea de *Runnable* pentru a da loc altui thread să se execute



# Prioritățile threadurilor și yield()

- Nu toate specificațiile JVM cer implementarea secționării temporale
  - Deși multe JVM folosesc secționarea temporală, unele folosesc un planificator de threaduri ce permite unui thread să execute în întregime metoda `run()`
- Nu vă bazați pe prioritățile threadurilor când proiectați o aplicație multithreading, deoarece comportamentul de planificare bazat pe priorități nu este garantat
  - Ex.: dacă un thread intră în starea *runnable*, dar are prioritate mai mare decât threadul curent care rulează, atunci threadul curent este trimis în starea *runnable* și thread-ul cu prioritatea cea mai mare este ales să ruleze
- Folosiți prioritățile threadurilor ca o modalitate de a îmbunătăți eficiența programului



# Setarea priorității unui thread

- Un thread primește implicit o prioritate în momentul în care este creat
  - Ex.: `MyThread t = new MyThread(); // prioritatea implicită are valoarea 5`
  - Deoarece threadul **main** se execută în momentul în care este instantiat threadul **t**, acesta va avea aceeași prioritate ca **main**
- Setarea manuală a priorității unui thread se face folosind metoda `setPriority(...)` astfel:

```
MyThread t = new MyThread();
```

```
t.setPriority(8);
```

```
t.start();
```

- Valorile priorităților sunt de obicei între 1 (minimă) și 10 (maximă)
- Pentru a verifica exact intervalul permis de JVM folosiți constantele: `Thread.MIN_PRIORITY`, `Thread.NORM_PRIORITY`, `Thread.MAX_PRIORITY`



# Metoda statică Thread.yield()

- Metoda `yield()` are rolul de a duce threadul curent din stadiul de *running* în *runnable* pentru a permite threadurilor cu **aceeași prioritate** să ruleze și ele
- În realitate însă, metoda `yield()` nu garantează acest lucru
  - Chiar dacă `yield()` cauzează trecerea threadului curent în starea *runnable*, se întâmplă adesea ca același thread să fie ales spre execuție



# Metoda non-statică Thread.join()

- Metoda **join()** permite unui thread să se poziționeze la sfârșitul altui thread ("*join onto the end*")
- Dacă un thread B nu poate să își facă treaba decât după ce threadul A a fost executat, atunci vrem să se facă "join" între threadurile A și B
  - Acest lucru înseamnă că threadul B nu devine *Runnable* decât după ce threadul A a fost executat
- Ex.: `Thread t = new Thread();  
t.start();  
t.join(); // sau t.join(500);`
  - Acest cod ia threadul în lucru (în acest caz - main) și face "join" cu threadul t
  - Metoda **join()** cu parametru se interpretează astfel: threadul principal așteaptă până când t este finalizat, dar dacă durează mai mult de 500 ms, acesta devine din nou *Runnable*



# Sincronizarea codului

- Sincronizarea metodelor previne accesarea simultană a codului dintr-o metodă de mai multe threaduri
  - Ce s-ar întâmpla dacă două threaduri ar încerca simultan să seteze starea unui obiect?
- Cuvântul cheie **synchronized** poate fi folosit
  - Ca modifier al metodei
  - La începutul unui bloc de cod
- În timp ce un singur thread are dreptul de a accesa codul sincronizat al unei instanțe, restul codului (nesincronizat) poate fi accesat simultan de mai multe threaduri
- Când un thread intră în starea *sleep* codul sincronizat blocat de el nu va fi accesibil altor threaduri



# Sincronizarea codului. Exemplu

- Un cont bancar cu două persoane împărtășite să aibă acces la acest cont

```
class Account {  
    private int balance = 50;  
    public int getBalance() {  
        return balance;  
    }  
    public void withdraw(int amount) {  
        balance = balance - amount;  
    }  
}
```

- Din cont se pot efectua operații de retragere de numerar. Acestea sunt limitate la suma exactă de 10. Astfel, pașii de efectuat ar fi:
  - Verifică bilanțul
  - Dacă sunt suficienți bani (ex. minim 10), efectuează retragerea



# Sincronizarea codului. Exemplu

- Ce se întâmplă dacă intervene ceva între acești pași?
  - Ex.: Cei doi utilizatori vor să retragă bani în același timp, pentru amândoi bilanțul arată că este credit suficient, primul retrage suma, dar când al doilea încearcă să retragă, nu mai are bani suficienți în cont!
- Logica ce ar trebui urmată pentru implementarea problemei:
  1. Obiectul runnable să țină o referință către un singur cont
  2. Se pornesc două threaduri reprezentând acțiunile celor două persoane, și ambele threaduri fac referire la același obiect runnable
  3. Bilanțul initial e de 50 și suma de scos e exact 10
  4. În metoda run() avem un ciclu ce se repetă de 5 ori. În fiecare buclă
    - Efectuăm retragerea (doar în cazul în care există destul credit)
    - Afisăm un mesaj dacă contul este 'corupt' (lucru care nu ar trebui să se întâmple niciodată, pentru că verificăm de fiecare bilanțul contului)



# Sincronizarea codului. Exemplu

5. Metoda `makeWithdrawal()` într-o clasă de test ar trebui să efectueze următoarele:
  - Verifică bilanțul pentru a verifica dacă este suficient credit pentru o retragere
  - Dacă este suficient, afișează numele persoanei care operează contul
  - Pune threadul pe *sleep* 500ms, simulând timpul necesar pentru o retragere
  - La trezire, efectuează retragerea și afișează un mesaj în acest sens
  - Dacă nu au fost bani suficienți, afișează un mesaj care să atenționeze asupra acestui lucru



# Sincronizarea codului. Exemplu

```
public class AccountDanger implements Runnable {  
    private Account acct = new Account();  
    public static void main (String [] args) {  
        AccountDanger r = new AccountDanger();  
        Thread one = new Thread(r);  
        Thread two = new Thread(r);  
        one.setName("Fred");  
        two.setName("Lucy");  
        one.start();  
        two.start();  
    }  
    public void run() {  
        for (int x = 0; x < 5; x++) {  
            makeWithdrawal(10);  
            if (acct.getBalance() < 0) {  
                System.out.println("account is overdrawn!");  
            }  
        }  
    }  
}
```

```
private void makeWithdrawal(int amt) {  
    if (acct.getBalance() >= amt) {  
        System.out.println(Thread.currentThread().  
                           getName() + " is going to withdraw");  
        try {  
            Thread.sleep(500);  
        } catch(InterruptedException ex) { }  
        acct.withdraw(amt);  
        System.out.println(Thread.currentThread().  
                           getName() + " completes the withdrawal");  
    } else {  
        System.out.println("Not enough in account for "  
                           + Thread.currentThread().getName()  
                           + " to withdraw " + acct.getBalance());  
    }  
}
```



# Sincronizarea codului. Exemplu

## ■ Rezultate afisate:

1. Fred is going to withdraw
2. Lucy is going to withdraw
3. Fred completes the withdrawal
4. Fred is going to withdraw
5. Lucy completes the withdrawal
6. Lucy is going to withdraw
7. Fred completes the withdrawal
8. Fred is going to withdraw
9. Lucy completes the withdrawal
10. Lucy is going to withdraw
11. Fred completes the withdrawal
12. Not enough in account for Fred to withdraw 0
13. Not enough in account for Fred to withdraw 0
14. Lucy completes the withdrawal
15. account is overdrawn!
16. Not enough in account for Lucy to withdraw -10
17. account is overdrawn!
18. Not enough in account for Lucy to withdraw -10
19. account is overdrawn!

Soluție pentru evitarea coruperii contului:  
Să ne sigurăm că pașii  
1. de verificare a bilanțului și  
2. de retragere se efectuează împreună  
(= operație atomică)



# Sincronizarea codului. Exemplu

- Nu se poate garanta că un singur thread va rula până la finalizarea operației atomice, dar putem garanta că, chiar dacă threadul își mai schimbă starea din modul running, nici un alt thread nu va putea să opereze cu aceste date
  - Cu alte cuvinte, chiar dacă threadul lui Lucy intră în modul sleep după verificarea bilanțului, putem să îi interzicem lui Fred să verifice și el bilanțul până când Lucy finalizează operația de retragere
- Cum protejăm datele?
  - Facem variabilele *private*
  - Sincronizăm codul care modifică variabilele
- Putem rezolva problema punând modificatorul **synchronized** metodei **makeWithdrawal()** astfel:

```
private synchronized void makeWithdrawal(int amt) {  
    //same code  
}
```



# Sincronizarea codului. Exemplu

## ■ Noile rezultate afisate:

Fred is going to withdraw

Fred completes the withdrawal

Lucy is going to withdraw

Lucy completes the withdrawal

Fred is going to withdraw

Fred completes the withdrawal

Lucy is going to withdraw

Lucy completes the withdrawal

Fred is going to withdraw

Fred completes the withdrawal

Not enough in account for Lucy to withdraw 0

Not enough in account for Fred to withdraw 0

Not enough in account for Lucy to withdraw 0

Not enough in account for Fred to withdraw 0

Not enough in account for Lucy to withdraw 0



# Sincronizare și blocare (*lock*)

- Sincronizarea funcționează cu lacăte (*locks*)
- Fiecare obiect Java are încorporat un singur lock care intervine doar atunci când obiectul deține metode sincronizate
- Doar metode (sau blocuri) pot fi sincronizate, nu și variabilele sau clasele
- Nu toate metodele necesită să fie sincronizate
- O clasă poate avea atât metode sincronizate, cât și nesincronizate
- Dacă două metode încearcă să execute o metodă sincronizată, și ambele folosesc aceeași instanță, threadurile vor putea să execute doar pe rând această metodă



# Interacțiunea dintre threaduri

- Ultimul lucru de știut despre threaduri este cum interacționează între ele pentru a-și comunica diferite aspecte (ex. *lock status*)
- Metodele responsabile pentru comunicarea între threaduri sunt: `wait()`, `notify()`, `notifyAll()` din clasa `Object`
- Un exemplu ar fi o aplicație de mail:
  - Aplicația are două threaduri: unul responsabil cu trimitera mailurilor (T1), iar celălalt cu procesarea lor (T2)
  - T2 trebuie să verifice frecvent dacă dacă există vreun mail de procesat
  - Folosind ***mecanismul wait-notify***, T2 poate verifica existența mailurilor de procesat, și dacă nu găsește nici unul, poate spune astfel: "nu o să-mi irosesc timpul verificând în continuu, ies să fac altceva, iar când T1 (mail delivery) aduce un mail, îmi va trimite o notificare să știu să trec în starea *Runnable* și să îmi fac treaba"



# Comunicarea cu obiectele cu metodele `wait()` și `notify()`

- Metoda `wait()` lasă threadul să spună: nu este nimic de lucru pentru mine asa ca pune-mă în starea *waiting*, sau adaugămă în lista de așteptare
- Metoda `notify()` este folosită pentru a transmite un semnal către un thread din lista de așteptare a obiectului
  - Nu se poate specifica pe care *waiting* thread să îl notifice
- Metoda `notifyAll()` trimite semnal tuturor threadurilor în starea *waiting*
- Metodele `wait()`, `notify()`, `notifyAll()` trebuie să fie apelate dintr-un cod sincronizat!
  - Un thread invocă una din aceste metode pe un obiect anume și threadul trebuie să dețină lock-ul aceluiași obiect



# Exemplu: Comunicarea cu obiectele cu metodele `wait()` și `notify()`

```
1. class ThreadA {  
2.     public static void main(String [] args) {  
3.         ThreadB b = new ThreadB();  
4.         b.start();  
5.  
6.         synchronized(b) {  
7.             try {  
8.                 System.out.println("Waiting for b to  
9.                     complete...");  
10.            b.wait();  
11.        } catch (InterruptedException e) {}  
12.        System.out.println("Total is: " +  
13.                           b.total);  
14.    }  
15.}
```

```
16. class ThreadB extends Thread {  
17.     int total;  
18.  
19.     public void run() {  
20.         synchronized(this) {  
21.             for(int i=0; i<100; i++) {  
22.                 total += i;  
23.             }  
24.             notify();  
25.         }  
26.     }  
27. }
```



# Exemplu: Comunicarea cu obiectele cu metodele `wait()` și `notify()`

- Explicații:
  - Programul conține două threaduri:
    - ThreadA – threadul principal
    - ThreadB – threadul care calculează suma numerelor de la 0 la 99
  - Linia 4: la apelul metodei `start()`, ThreadA continuă cu linia următoare de cod din clasa lui, ceea ce înseamnă că ar putea ajunge la linia 11 înainte ca ThreadB să finalizeze de calculat suma. Pentru a preveni acest lucru folosim metoda `wait()` la linia 9
  - La linia 6 codul este sincronizat cu obiectul b deoarece, pentru a putea apela metoda `wait()` pentru obiectul b, ThreadA trebuie să dețină lock-ul obiectului b
    - Pentru ca un thread să poată apela metodele `wait()` sau `notify()`, acesta trebuie să dețină lock-ul obiectului
    - Când un thread așteaptă (este în starea *wait*), el cedează pentru moment lock-ul obiectului în favoarea altui thread



# Exemplu: Comunicarea cu obiectele cu metodele `wait()` și `notify()`

- Remarcați la liniile 7-10 folosirea mecanismului try-catch în jurul metodei `wait()`
  - Un thread poate fi întrerupt prin apelul metodei `wait()` care aruncă o excepție. Astfel, excepția aruncată trebuie prinsă și tratată
  - ```
try {
    wait();
} catch(InterruptedException e) {
    // Do something about it
}
```
  - Odată intrat în starea `waiting`, threadul așteaptă până când operatorul trimite prima notificare, moment în care intră din nou în posesia lock-ului și își poate continua execuția
- Rezultatele afișate de programul din exemplu sunt:

Waiting for b to complete...

Total is: 4950



# Animații cu threaduri

- Animațiile sunt o sarcină uzuală pentru firele de lucru
- Firele de lucru
  - Pot efectua sarcini diferite în paralel
  - Sunt folosite la controlul animației
- Exemplu:
  - Un fir de lucru: realizează desenarea fiecărui cadru
  - Alt fir de lucru: tratează interacțiunile cu utilizatorul